Operating Systems 200 Assignment

Alex Eric Burress

14571008

19/05/2014

Assumptions

1. The SJF implementation is non-pre-emptive.
2. The text file containing process arrival and burst times contains one line at the top containing a single integer representing the time quantum, and subsequent lines of integers representing arrival and burst times of a single process per line.
3. Process arrival and burst times are either separated by a single space or tab.
4. Processes are listed in the text file with arrival times in ascending order.
5. No executable is expected to deal with incorrectly formatted text files.

List of Programs

1. **CompareSchedulers**
2. **RRScheduler**
3. **SJFScheduler**

CompareSchedulers implements pthreads. It prompts the user to enter the filename of a text file containing a time quantum and list of process arrival and burst times in the current directory. It then allows 2 threads to pull the data from the text file and calculate RR and non-pre-emptive SJF average turnaround and wait times. The parent thread then outputs the results to the user and the user has the option of running more simulations on different files.

RRScheduler does not implement pthreads. It prompts the user to enter the filename of a text file containing a time quantum and list of process arrival and burst times in the current directory. It then calculates and outputs the average turnaround and wait times for a RR scheduler. The user can run simulations on another file or quit.

SJFScheduler does not implement pthreads. It prompts the user to enter the filename of a text file containing a time quantum and list of process arrival and burst times in the current directory. It then calculates and outputs the average turnaround and wait times for a non-pre-emptive SJF scheduler. The user can run simulations on another file or quit. SJFScheduler scans in a time quantum but does nothing with the value.

Compilation and Execution

A makefile is supplied. The command “make all” will compile all three executables. They can also be compiled separately with the following:

- make CompareSchedulers
- make RRScheduler
- make SJFScheduler

The executables can be run using “./”, for example CompareSchedulers can be run using:

```
./CompareSchedulers
```

Mutual Exclusion

Mutual exclusion is required for the CompareSchedulers executable. There are three threads that run simultaneously while accessing or modifying global variables. The global variables are named “userInput” and “stats”. userInput is the string holding the filename the user enters and stats is a struct that holds average turnaround and wait times for a batch of processes scheduled via RR or non-pre-emptive scheduling. Both variables are placed in buffers and can only be accessed by certain threads when their buffers meet the right conditions.

```
/* buffer1 is used to hold the filename */
int buffer1 = 0;
char userInput[10];

/* buffer2 is used to hold the average turnaround and wait times */
int buffer2 = 1;
ProcessStats stats;
```

The three threads that access these global variables are “parent”, “rr” and “sjf”.

```
pthread_t parent;
pthread_t rr;
pthread_t sjf;
```

Mutual exclusion for userInput is achieved by the following rules:

1. If parent wants to modify the filename, it will wait for a signal from sjf showing that sjf has successfully used the previous filename.
2. Once parent modifies the filename, it sends a signal to rr. Thread rr then receives the signal and opens the file. Once rr is is done with the file it signals thread sjf that the file is ready.
3. Thread sjf then reads the file and signals parent that the filename can be changed.

Mutual exclusion for stats is achieved by the following rules:

1. Parent must wait for a signal from thread sjf communicating that sjf has updated the stats global variable before outputting results to the user.
2. Thread rr must wait for a signal from parent communicating that parent is awaiting new times.
3. Thread sjf must wait for a signal from thread rr communicating that rr has successfully modified the stats variable.

Bugs

1. Valgrind shows that 408 bytes of memory are possibly lost each time the CompareSchedulers executable is run.
2. Average times are listed as “nan” if the text file contains lines with non-numeric values.
3. Average times are listed as “nan” if arrival, burst and time quantum values are listed as floats.

Sample Inputs and Outputs

Sample 1:

```
2
0 7
1 3
5 2
7 2
8 4
```

Output for sample 1:

```
Enter filename: t.txt
RR:     avg turnaround = 8.400000, avg wait = 4.800000
SJF:    avg turnaround = 7.600000, avg wait = 4.000000
```

Output is correct.

Sample 2:

```
8
0   26
0   2
1   1
5   36
5   4
8   9
10  52
```

Output for sample 2:

```
Enter filename: a.txt
RR:     avg turnaround = 56.142857, avg wait = 37.571429
SJF:    avg turnaround = 37.714286, avg wait = 19.142857
```

Output is correct.

Source Code

Note: Source code for RRScheduler and SJFScheduler have been omitted because the source code is very similar to the RR and SJF scheduling code in CompareSchedulers. The only difference is that CompareSchedulers contains communication between threads and handling of critical sections, whereas RRScheduler and SJFScheduler don't require these features.

CompareSchedulers.c

```
/**
 * OS200 2014 Assignment
 * Author: Alex Burress
 * Student ID: 14571008
 *
 * This program creates three threads: parent, rr and sjf. These three
threads
 * work together to calculate the average turnaround and wait times for a
list
 * of processes in a text file. The user is prompted for the name of a text
file
 * in the current directory containing process arrival and burst times. The
rr
 * thread calculates times using a round robin scheduling algorithm, at the
same
 * time the sjf thread calculates times for a non preemptible shortest job
 * first scheduler. These values are passed back to the parent thread which
 * outputs the results to the user. The program will exit if the user
provides
 * "QUIT" or "quit" as a filename, or if the user enters an invalid
filename, or
 * if the file does not open.
 *
 * All three threads will run continuously until an error is encountered or
the
 * elects to quit.
 */

#include "CompareSchedulers.h"
#define FALSE 0
#define TRUE !FALSE

/* global variables used for threads and thread communication */
pthread_t parent;
pthread_t rr;
pthread_t sjf;
pthread_mutex_t modifyFilename;
pthread_mutex_t modifyStats;
pthread_cond_t noFilename;
pthread_cond_t rrScheduled;
pthread_cond_t sjfScheduled;
pthread_cond_t rrValsEntered;
pthread_cond_t sjfValsEntered;
pthread_cond_t noValsEntered;


/* buffer1 is used to hold the filename */
int buffer1 = 0;
char userInput[10];

/* buffer2 is used to hold the average turnaround and wait times */
int buffer2 = 1;
ProcessStats stats;

int main()
{
    /* initialise mutexes and conditions */
    pthread_mutex_init(&modifyFilename, NULL);
    pthread_mutex_init(&modifyStats, NULL);
    pthread_cond_init(&noFilename, NULL);
    pthread_cond_init(&rrScheduled, NULL);
    pthread_cond_init(&sjfScheduled, NULL);
    pthread_cond_init(&rrValsEntered, NULL);
    pthread_cond_init(&sjfValsEntered, NULL);
    pthread_cond_init(&noValsEntered, NULL);

    /* create all three pthreads */
    pthread_create(&parent, NULL, runPrompt, NULL);
    pthread_create(&rr, NULL, runRR, NULL);
    pthread_create(&sjf, NULL, runSJF, NULL);

    /* join threads */
    pthread_join(parent, NULL);
    pthread_join(rr, NULL);
    pthread_join(sjf, NULL);

    return 0;
}

/**
 * runPrompt()runs continuously and acts as the parent thread. It will
prompt
 * the user for a filename at program startup and after the rr and sjf
 * threads have successfully taken the filename from buffer1. It also
prints
 * the times calculated by the rr and sjf threads.
 */

void* runPrompt()
{
    while (TRUE)
    {
        /* wait for the filename buffer to be empty */
        pthread_mutex_lock(&modifyFilename);
        while (buffer1 != 0)
        {
            pthread_cond_wait(&noFilename, &modifyFilename);
        }

        /* scan filename into userInput */
        buffer1 = 1;
        printf("Enter filename: ");
        scanf("%s", userInput);

        /* if user entered "quit" exit the program */
        if (strcmp(userInput, "quit") == 0 || strcmp(userInput, "QUIT") ==
0)
        {
            exitProgram();
        }
        pthread_cond_signal(&rrScheduled);
        pthread_mutex_unlock(&modifyFilename);

        /* wait for time values from schedulers */
        pthread_mutex_lock(&modifyStats);
        while (buffer2 != 0)
        {
            pthread_cond_wait(&sjfValsEntered, &modifyStats);
        }
```

```c
        /* print schedule times */
        buffer2 = 1;
        printf("RR:\tavg turnaround = %f, avg wait = %f\n",
stats.rrTurnaround, stats.rrWait);
        printf("SJF:\tavg turnaround = %f, avg wait = %f\n",
stats.sjfTurnaround, stats.sjfWait);

        pthread_cond_signal(&noValsEntered);
        pthread_mutex_unlock(&modifyStats);
    }
    return NULL;
}

/**
 * runRR() runs continuously and acts as the round robin scheduler. It
waits for
 * the parent thread to produce a filename for buffer1, then runs the RR
 * scheduling algorithm for the file. Once scheduling has finished, it
updates
 * buffer2 with the RR scheduler times.
 */

void* runRR()
{
    LinkedList* arriving;
    LinkedList* finished;
    int timeQuantum;
    double avgWait;
    double avgTurnaround;

    while (TRUE)
    {
        /* wait for a filename to go into buffer1 */
        pthread_mutex_lock(&modifyFilename);
        while (buffer1 != 1)
        {
            pthread_cond_wait(&rrScheduled, &modifyFilename);
        }

        /* pull values from text file */
        buffer1 = 2;
        arriving = createList();
        timeQuantum = pullScheduleDataRR(arriving);

        pthread_cond_signal(&sjfScheduled);
        pthread_mutex_unlock(&modifyFilename);

        /* if the file has a valid time quantum, run RR scheduler */
        if (timeQuantum > 0)
        {
            /* find average wait and turnaround times */
            finished = runTimingLoopRR(arriving, timeQuantum);
            calcTimes(finished, &avgWait, &avgTurnaround);

            /* wait for parent to print previous stats */
            pthread_mutex_lock(&modifyStats);
            while (buffer2 != 1)
            {
                pthread_cond_wait(&noValsEntered, &modifyStats);
            }
            /* update RR times */
            buffer2 = 2;
            stats.rrTurnaround = avgTurnaround;
            stats.rrWait = avgWait;

            pthread_cond_signal(&rrValsEntered);
            pthread_mutex_unlock(&modifyStats);
        }
    }

    return NULL;
}

/**
 * runSJF() runs continuously and acts as the non-preemptible SJF
scheduler. It
 * waits for the RR thread to open and close the text file, then pulls
value
 */

void* runSJF()
{
    LinkedList* jobs;
    LinkedList* finished;
    double avgWait;
    double avgTurnaround;

    while (TRUE)
    {
        /* wait for the rr thread to close the text file */
        pthread_mutex_lock(&modifyFilename);
        while (buffer1 != 2)
        {
            pthread_cond_wait(&sjfScheduled, &modifyFilename);
        }
        buffer1 = 0;

        /* pull values from the text file */
        jobs = createList();
        pullScheduleDataSJF(jobs);

        pthread_cond_signal(&noFilename);
        pthread_mutex_unlock(&modifyFilename);

        /* run SJF scheduler */
        finished = runTimingLoopSJF(jobs);
        calcTimes(finished, &avgWait, &avgTurnaround);

        /* wait for rr thread to update RR times */
        pthread_mutex_lock(&modifyStats);
        while (buffer2 != 2)
        {
            pthread_cond_wait(&rrValsEntered, &modifyStats);
        }

        /* update buffer2 with SJF times */
        buffer2 = 0;
        stats.sjfTurnaround = avgTurnaround;
        stats.sjfWait = avgWait;

        pthread_cond_signal(&sjfValsEntered);
        pthread_mutex_unlock(&modifyStats);

        freeList(jobs);
    }
    return NULL;
}

/**
 * pullScheduleDataRR() pulls a time quantum value as well as burst and
 * arrival times for every process in the text file. The time quantum is
 * returned while the arrival and burst times are queued in the passed in
 * linked list.
 */

int pullScheduleDataRR(LinkedList* list)
{
    FILE* source;
    int timeQuantum;
    int arrivalTime;
    int burstTime;
    Process* newProcess;

    source = fopen(userInput, "r");

    /* if file didn't open, exit program */
    if (source == NULL)
    {
        printf("Error opening file, exiting.\n");
        freeList(list);
        exitProgram();
    }
    else
    {
        /* retrieve time quantum */
        fscanf(source, "%d", &timeQuantum);

        /* retrieve arrival and burst times for each process and enqueue */
        while(fscanf(source, "%d %d", &arrivalTime, &burstTime) == 2)
        {
            newProcess = (Process*)malloc(sizeof(Process));
            newProcess->arrivalTime = arrivalTime;
            newProcess->burstTime = burstTime;
            newProcess->remainingBurst = burstTime;
            newProcess->waitTime = 0;
            newProcess->lastAllocated = -1;
            enqueue(list, newProcess);
        }

        fclose(source);
    }

    return timeQuantum;
}

/**
 * runTimingLoopRR() simulates RR scheduling for processes in the passed in
 * linked list "arriving". It runs a timing loop where at every time
quantum it
 * places newly arrived processes on the ready queue and simulates
dispatching
 * processes to the CPU if the CPU is free. It will preempt processes in
```

```
the CPU
 * if the time quantum has expired.
 */

LinkedList* runTimingLoopRR(LinkedList* arriving, int timeQuantum)
{
    int totalProcesses;
    int currentTime;
    int processorBusy;
    int partialTimeSlice;
    Process* peekedProcess;
    Process* readyProcess;
    Process* allocatedProcess;
    LinkedList* ready;
    LinkedList* finished;

    /* initialise variables and set current time to 0 */
    ready = createList();
    finished = createList();
    currentTime = 0;
    processorBusy = FALSE;
    totalProcesses = arriving->count;

    /* simulate CPU scheduling until no jobs are left in ready or arriving
queues */
    while(finished->count < totalProcesses)
    {
        /* transfer processes that have arrived to the ready queue */
        while (arriving->count > 0 && currentTime >=
retrieveElement(arriving, 0)->arrivalTime)
        {
            peekedProcess = retrieveElement(arriving, 0);
            if (peekedProcess->arrivalTime <= currentTime)
            {
                readyProcess = dequeue(arriving);
                enqueue(ready, readyProcess);
            }
        }

        /* If CPU is busy, check on the status of the process in the CPU */
        if (processorBusy == TRUE)
        {
            partialTimeSlice++;
            allocatedProcess->remainingBurst--;

            /* free up the CPU if the current process finished */
            if (allocatedProcess->remainingBurst <= 0)
            {
                allocatedProcess->turnaroundTime = currentTime -
allocatedProcess->arrivalTime;
                allocatedProcess->waitTime = currentTime -
allocatedProcess->burstTime - allocatedProcess->arrivalTime;
                enqueue(finished,allocatedProcess);
                processorBusy = FALSE;
            }

            /* preempt the process in the CPU if timeslice expired */
            else if (partialTimeSlice == timeQuantum)
            {
                enqueue(ready, allocatedProcess);
                processorBusy = FALSE;
            }
        }

        /* dispatch first job in the ready queue to the CPU if CPU isn't
busy */
        if (processorBusy == FALSE && ready->count > 0)
        {
            processorBusy = TRUE;
            allocatedProcess = dequeue(ready);
            partialTimeSlice = 0;
        }

        /* increment time by one time quantum */
        currentTime++;
    }

    freeList(ready);
    freeList(arriving);

    return finished;
}

/**
 * pullScheduleDataSJF() works the same as pullScheduleDataRR(), but
doesn't
 * return a time quantum value.
 */

void pullScheduleDataSJF(LinkedList* list)
{
    FILE *source;
    int timeQuantum;
    int arrivalTime;
    int burstTime;
    Process* newProcess;

    /* try to open file */
    source = fopen(userInput, "r");

    /* quit program if file wouldn't open */
    if (source == NULL)
    {
        printf("Error opening file, exiting.\n");
        freeList(list);
        exitProgram();
    }
    else
    {
        /* scan time quantum and put aside */
        fscanf(source, "%d", &timeQuantum);

        /* enqueue all processes listed in the text file */
        while(fscanf(source, "%d %d", &arrivalTime, &burstTime) == 2)
        {
            newProcess = (Process*)malloc(sizeof(Process));
            newProcess->arrivalTime = arrivalTime;
            newProcess->burstTime = burstTime;
            newProcess->waitTime = 0;
            enqueue(list, newProcess);
        }
        fclose(source);
    }
}

/**
 * runTimingLoopSJF() simulates CPU scheduling for a non-preemptible SJF
scheme.
 * It dispatches the first job in the ready queue to the CPU, if no job is
in
 * the ready queue it checks whether more are due to arrive. If more
processes
 * are on the way it increments the timer until the next job arrives, then
 * dispatches it. Stats for finished processes are returned in a linked
list.
 */

LinkedList* runTimingLoopSJF(LinkedList* jobs)
{
    int currentTime;
    int mustWait;
    int searchedAll;
    int foundJob;
    int ii;
    Process* candidate;
    Process* shortestJob;
    LinkedList* finished;

    /* set time to 0 to begin with */
    currentTime = 0;
    foundJob = FALSE;
    finished = createList();

    while (jobs->count > 0)
    {
        mustWait = TRUE;
        searchedAll = FALSE;

        /* see if any eligible jobs are in the queue */
        while (mustWait == TRUE && searchedAll == FALSE)
        {
            for (ii = 0; ii < jobs->count; ii++)
            {
                candidate = retrieveElement(jobs, ii);
                if (candidate->arrivalTime <= currentTime)
                {
                    shortestJob = candidate;
                    mustWait = FALSE;
                }
                if (ii == (jobs->count - 1))
                {
                    searchedAll = TRUE;
                }
            }
        }

        /* at least 1 eligible job exists */
        if (mustWait == FALSE)
        {
            shortestJob = dequeue(jobs);
            for (ii = 1; ii < jobs->count; ii++)
            {
                candidate = dequeue(jobs);
                if (candidate->arrivalTime <= currentTime && candidate-
>burstTime < shortestJob->burstTime)
                {
                    enqueue(jobs, shortestJob);
                    shortestJob = candidate;
                }
                else
                {
                    enqueue(jobs, candidate);
                }
            }

            /* save the wait time for the dispatched job */
            shortestJob->waitTime = currentTime - shortestJob->arrivalTime;

            /* simulate CPU time */
            currentTime += shortestJob->burstTime;

            /* after CPU has finished, save the turnaround time for the job
*/
            shortestJob->turnaroundTime = currentTime - shortestJob-
>arrivalTime;
            enqueue(finished, shortestJob);
        }

        /* no eligible job exists, increment time */
        if (searchedAll == FALSE)
        {
            currentTime++;
        }
    }

    return finished;
}

/**
 * calcTimes() calculates the average turnaround and wait times for a
linked
 * list of finished processes.
 */

void calcTimes(LinkedList* finished, double* avgWait, double*
avgTurnaround)
{
    int totalWait;
    int totalTurnaround;
    int countProcesses;
    Process* finishedProcess;

    totalWait = 0;
    totalTurnaround = 0;
    countProcesses = finished->count;

    /* run through the list and tally the total wait and turnaround times
*/
    while (finished->count > 0)
    {
        finishedProcess = dequeue(finished);
        totalWait += finishedProcess->waitTime;
        totalTurnaround += finishedProcess->turnaroundTime;
```

```c
        free(finishedProcess);
    }

    freeList(finished);

    /* calculate the average times */
    *avgWait = (double)totalWait/(double)countProcesses;
    *avgTurnaround = (double)totalTurnaround/(double)countProcesses;
}

/**
 * exitProgram() will destroy all mutexes and conditions and exit the
program.
 */

void exitProgram()
{
    pthread_mutex_destroy(&modifyFilename);
    pthread_mutex_destroy(&modifyStats);
    pthread_cond_destroy(&noFilename);
    pthread_cond_destroy(&rrScheduled);
    pthread_cond_destroy(&sjfScheduled);
    pthread_cond_destroy(&rrValsEntered);
    pthread_cond_destroy(&sjfValsEntered);
    pthread_cond_destroy(&noValsEntered);

    exit(0);
}
```

LinkedList.c

```
/**
 * A set of functions for creating and manipulating a linked list of Event 
structs.
 * Each linked list has a head and a tail pointer, and a count of nodes on 
the list.
 *
 * Author: Alex Burress
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>
#include "LinkedList.h"

/**
 * Creates an empty linked list.
 */
LinkedList* createList()
{
    LinkedList* newList;
    newList = (LinkedList*)malloc(sizeof(LinkedList));
    newList->head = NULL;
    newList->tail = NULL;
    newList->count = 0;

    return newList;
}

/**
 * Creates a node for the passed-in Process and inserts the node at the 
start of
 * the list.
 */
void insertFirst(LinkedList* list, Process* process)
{
    ListNode* newNode;
    ListNode* current;
    int ii;

    newNode = (ListNode*)malloc(sizeof(ListNode));
    newNode->data = process;

    /* if list is empty, point head and tail at the new node */
    if (list->head == NULL)
    {
        list->head = newNode;
        newNode->next = NULL;
        list->tail = newNode;
    }
    /* else make the new node point at the first node, and have head point 
to
     * the new node.
     */
    else
    {
        newNode->next = list->head;
        list->head = newNode;
        /* pointing tail to last node */
        current = list->head;
        for (ii = 1; ii < (list->count); ii++)
        {
            current = current->next;
        }
        list->tail = current;
    }

    list->count++;
}

/**
 * Creates a node for the passed-in Process, and inserts the node at the
end
 * of the list.
 */
void insertLast(LinkedList* list, Process* process)
{
    ListNode* newNode;

    newNode = (ListNode*)malloc(sizeof(ListNode));
    newNode->data = process;
    newNode->next = NULL;

    /* if list is empty, point head and tail to the new node */
    if (list->count <= 0)
    {
        list->head = newNode;
        list->tail = newNode;
    }
    else
    {
        list->tail->next = newNode;
        list->tail = newNode;
    }
    list->count++;
}

/**
 * Inserts an element at the end of the list. Is a wrapper for insertLast()
and is
 * used when the list represents an unbounded queue.
 */

void enqueue(LinkedList* list, Process* process)
{
    insertLast(list, process);
}

/**
 * Removes the first node from the list, extracts and returns a pointer to
 * its Process, and frees the leftover node. Returns NULL if the passed-in
list
 * has no nodes.
 */
Process* removeFirst(LinkedList* list)
{
    ListNode* removedNode;
    Process* outProcess;
    /* If list is empty, print error message */
    if (list->count <= 0)
    {
        printf("Error: linked list is empty\n");
        removedNode = NULL;
        outProcess = NULL;
        list->count = 0;
    }
    /* if list has only one node */
    else if (list->count == 1)
    {
        removedNode = list->head;
        list->head = NULL;
        list->tail = NULL;
        list->count--;
        outProcess = removedNode->data;
    }
    /* else list has 2+ nodes */
    else
    {
        removedNode = list->head;
        list->head = list->head->next;
        removedNode->next = NULL;
        list->count--;
        outProcess = removedNode->data;
    }

    free(removedNode);

    return outProcess;
}

/**
 * Removes the first element from the list, used as a wrapper for
removeFirst()
 * when using a linked list as an unbounded queue.
 */

Process* dequeue(LinkedList* list)
{
    return removeFirst(list);
}

/**
 * Deletes the n'th element on the linked list along with its node. The
 * passed-in int elNo represents n. Elements have a 0-based count.
 * i.e. if elNo = 0, the first node on the list is deleted. The variable
 * 'previous' is used to keep track of the node preceding to the node being
 * deleted. When the n'th node is deleted, previous->next is pointed to the
 * deleted node's next address.
 */
void deleteNthElement(LinkedList* list, int elNo)
{
    ListNode* current;
    ListNode* previous;
    int ii;

    assert(elNo >= 0);
    assert(elNo < list->count);

    current = list->head;
    previous = NULL;

    /* if deleting node that isn't head */
    if (elNo > 0)
    {
        for (ii = 0; ii < elNo; ii++)
        {
            previous = current;
            current = current->next;
        }

        previous->next = current->next;
    }
    /* else deleting head node */
    else
    {
        list->head = current->next;
    }

    list->count--;

    free(current->data);
    free(current);
}

/**
 * Returns a pointer to the n'th element on the list, where the count for n
 * is 0-based. i.e. if elNo = 0, the first event on the list is retrieved.
 */
Process* retrieveElement(LinkedList* list, int elNo)
{
    ListNode* current;
    int ii;
    Process* returnProcess;

    current = list->head;

    /* check if list is empty */
    if (list->count == 0)
    {
        printf("Error: list is empty.\n");
        returnProcess = NULL;
    }
    /* validate that elNo is within bounds */
    else if (elNo > (list->count - 1))
    {
        printf("Error: There are fewer than %d elements in this list.\n",
(elNo + 1));
        returnProcess = NULL;
    }
    else
    {
        /* if head is being retrieved */
        if (elNo == 0)
        {
            returnProcess = current->data;
        }
        /* else traverse list to desired element */
        else
        {
            for (ii = 0; ii < elNo; ii++)
```

```c
            {
                current = current->next;
            }
            returnProcess = current->data;
        }
    }

    return returnProcess;
}

/**
 * Frees each list node and Process from memory, and the list itself.
 */
void freeList(LinkedList* list)
{
    ListNode* current;
    ListNode* next;
    int ii;

    current = list->head;

    for (ii = 0; ii < (list->count); ii++)
    {
        next = current->next;
        free(current->data); /* */
        free(current);
        current = next;
    }
    free(list);
}
```