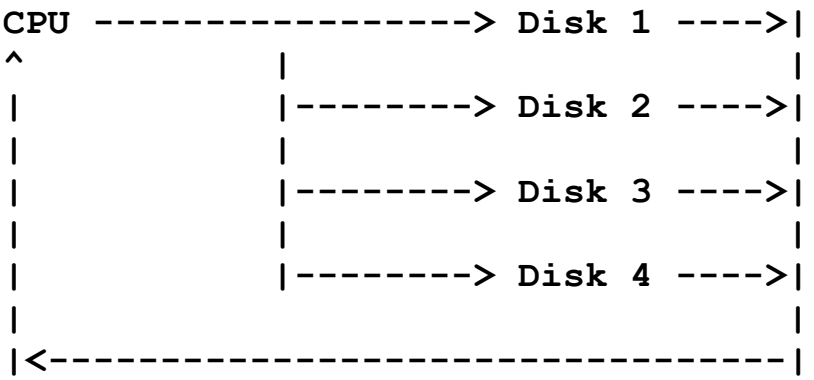
SimPack

CONTENTS
========

1. CPUDISK EXAMPLE

cpudisk represents the simulation of the following network composed of 1 CPU and 4 disks:



There are 9 jobs that move through the system. A single job alternately accesses CPU and a disk many times -- this simulates the effect of a real program using these types of resources. Of the 9 jobs, there are 6 low priority jobs and 3 high priority jobs.

2. SIMPACK QUEUING LIBRARY:

This is a set of routines that are functionally very similar to the routine library called SMPL by M. H. MacDougall ("Simulating Computer Systems: Techniques and Tools", MIT Press, 1987). SMPL was used a base, and this library contains a large number of extended options and additional routines.

Each routine will be listed followed by a description of its purpose.

2.1 Model Initialization

```
Usage: init_simpack(mode);
Type: int mode;
Example: init_simpack(HEAP);
```

This must be the first call to any of the SimPack routines. 'mode' can be one of 3 values: LINKED, HEAP or CALENDAR. If 'LINKED' is chosen, then a linear linked list is employed for the future event list.

If 'HEAP' is chosen, then a priority queue (i.e. heap) will be used instead. For relatively small future event lists, you will find that LINKED is perfectly acceptable. For longer lists, one should choose HEAP for the best performance. The 'CALENDAR' option is for very large future event lists. SUGGESTION: try all three options for your simulation to see which provides the best simulation times.

2.2 Facilities

```
** Usage: status(facility_id);
Type: int status(), facility_id;
Usage: if (status(cpu) == FREE) ...
```

Report back where the facility is BUSY or FREE.

```
** Usage: facility_id = create_facility(name, servers);
Type: int facility_id, servers; char *name;
```

Example: disk[2] = create_facility("disk", 1);

Create a resource (or "facility") called 'name' (in string form) which has 'servers' number of servers. This capability allows for single or multiple server facilities.

**** Usage: status = request(facility_id, token, priority);**
Example: if (request(cpu, token, 0) == FREE) ...

Token 'token' requests a facility. NOTE: tokens must be defined to be of type TOKEN. Look at the single server queue example under the 'func' subdirectory. The token has priority level 'priority'. The higher the value, the higher the priority. If the facility is free, then this token gets the facility (i.e. it is immediately served).

If the facility is currently busy then this token is placed with other tokens of the same priority level in the queue for this facility. 'status' is one of two values: FREE or BUSY. Implicit queuing is used in that the programmer need not explicitly queue the request if the facility is busy -- this is done automatically. NOTE: Use the first attribute of the token to store an integer token id:

```
TOKEN token;

token.attr[0] = 1.0;
```

Currently, tokens are structures with a number of floating point attributes.

This can easily be changed by looking at the definition of 'TOKEN' in the file 'queuing.c'. For many of your applications, you might require only a token identification; however, for some applications, you may want to store more than a token id when the item is put within the future event list (see the 'logic' program for an example).

**** Usage: status = preempt(facility_id,token,priority);**
Example: if (preempt(cpu,token,0) == FREE) ...

Token 'token' requests a facility. The token has priority level 'priority'. The higher the value, the higher the priority. If the facility is free, then this token gets the

facility (i.e. it is immediately served). Note that 'preempt' acts just like 'request' when the facility is not busy. If, on the other hand, the facility is busy then the current token's priority level is checked against the priority level of the token that currently has service. If it is less, then the preemptive token is placed in the queue (as in the 'request' call).

If, however, the preemptive token's priority is higher then the token preempts (i.e. replaces) the token that has current service by that facility/server. The preempted token is placed at the head of the facility queue and is re-started (for the amount of service time left) automatically once the preemptive token has finished service.

**** Usage: release(facility, token);**
Example: release(cpu, token);

The token 'token' releases the facility once it has had service.

WARNING: Always put a release in its own event routine (i.e., switch statement). Do NOT include it in the same event routine where you request another facility.

2.3 Time & Scheduling

**** Usage: time();**
Example: current_time = time();

Return the current simulation time.

**** Usage: busy_time(facility_id);**
Type: int facility_id;
Example: total_busy_time = busy_time(2);

Return the total amount of busy time for a facility.

**** Usage: schedule(event, inter-event-time, token);**
Example: schedule(2, 3.4, token);

Event number 'event' (for a token) is scheduled to occur at time =

current_time + inter-event-time

Note that 'inter-event-time' is always relative to the current simulation time.

**** Usage: next_event(&event, &token);**
Example: next_event(&event, &token);

Obtain the next event from the future event data structure. This procedure, in effect, 'causes' the event to occur.

**** Usage: int cancel_event(event);**
Example: token_num = cancel_event(event);

Look through the future event list and cancel a list item specified by 'event'. If this routine is successful in finding this event then 1) the corresponding token number is returned and 2) the linked list node is removed from the future event list. On the other hand, if the event cannot be found then the value NOT_FOUND (see queuing.h) is returned, and the future event list is left unchanged. NOTE: You must use LINKED for this routine.

**** Usage: int cancel_token(token);**
Example: event_num = cancel_token(token);

Look through the future event list and cancel a list item specified by 'token'. If this routine is successful in finding this token then 1) the corresponding event number is returned and 2) the linked list node is removed from the future event list. On the other hand, if the token cannot be found then the value NOT_FOUND (see queuing.h) is returned, and the future event list is left unchanged. NOTE: You must use LINKED for this routine.

2.4 Statistical Reporting

**** Usage: update_arrivals();**

Place in event code where tokens enter the system. This is used to keep track of the arrival rate.

**** Usage: update_completions();**

Place in event code where tokens leave the system. This is used to keep track of the completion rate (i.e. throughput).

**** Usage: report_stats();**

Provide a summary report on the facilities.

2.5 Tracing and Visualization

******SOME OF THESE FUNCTIONS DO NOT WORK ON DOS******

Usage: trace_visual(type);
Type: int type;

Place this after 'init_simpack' to specify the type of trace to do: either INTERACTIVE or BATCH. The interactive mode uses UNIX 'curses' to allow you to step through the simulation while viewing the key data structures (future event list, facility queues). The BATCH mode is useful to output this visual trace to a file. Currently, you must use LINKED (in init_simpack) to obtain a graphical trace.

Usage: trace_facility(facility_id);
Type: int facility_id;

Print the queue for the facility. This routine causes an immediate "dump" wherever it is placed within the code.

Usage: trace_eventlist();

Print the future event list if LINKED is used. This routine causes an immediate "dump" wherever it is placed within the code.

Usage: trace_eventlist_size();

Print the future event list size if LINKED is used. This routine causes an immediate "dump" wherever it is placed within the code.

Usage: print_heap();

Print out the heap (if HEAP is used).

2.6 Random Variates

NOTE: Made available originally from SMPL C-tool Library (M.H. MacDougall) with the exception of 'triang'.

** Usage: stream(stream_number);
Type: int stream_number;

Select a random number stream between 1 and 15.

** Usage: number = ranf();
Type: double ranf();

Return a pseudo random variate with a uniform distribution between 0 and 1.

** Usage: number = uniform(a, b);
Type: double uniform(a, b); double a, b;

Sample from a continuous uniform distribution in the range [a,b].

** Usage: number = random(i, j);
Type: int random(i, j); int i, j;

Sample from a discrete uniform distribution in the range i,i+1,...,j.

** Usage: number = expntl(x);
Type: double expntl(x); double x;

Sample from an exponential distribution with mean x.

**** Usage: number = normal(x, s);**
Type: double normal(x, s); double x, s;

Sample from a normal distribution with mean x and standard deviation s.

**** Usage: number = erlang(x, s);**
Type: double erlang(x, s); double x, s;

Sample from an Erlang distribution with mean x and standard deviation s.

**** Usage: number = hyperx(x, s);**
Type: double hyperx(x, s); double x, s;

Sample from a hyperexponential distribution with mean x and standard deviation s.

**** Usage: number = triang(a, c, b);**
Type: double triang(a, c, b),a, c, b;

Sample from a triangular distribution with endpoints a and b and mode c.