CS551
Warm-up Project #2
Bill Cheng

http://merlot.usc.edu/cs551-f12
Multi-threading Exercise

- Make sure you are familiar with the `pthreads` library
  - good source is the book by Nichols, Buttlar, and Farrell
  - you must learn how to use mutex and condition variables correctly
    - `pthread_mutex_lock()`/`pthread_mutex_unlock()`
    - `pthread_cond_wait()`/`pthread_cond_signal()`/`pthread_cond_broadcast()`
- you must learn how to handle UNIX signals
  - `pthread_sigmask()`/`sigwait()`
  - `pthread_setcancelstate()`
  - `pthread_setcanceltype()`
  - `pthread_testcancel()`
#include <pthread.h>

thread_t user_threadID;
sigset_t new;

void *handler(), interrupt();

main( int argc, char *argv[] )
{
    sigemptyset(&new);
sigaddset(&new, SIGINT);

    pthread_sigmask(SIG_BLOCK, &new, NULL);
    pthread_create(&user_threadID, NULL, handler, argv[1]);
    pthread_join(user_threadID, NULL);

    printf("thread handler, %d exited\n",user_threadID);
sleep(2);
    printf("main thread, %d is done\n", thr_self());
} /* end main */
Child thread example

- child thread unblocks SIGINT

```c
struct sigaction act;

void *
handler(char argv1[])
{
    act.sa_handler = interrupt;
    sigaction(SIGINT, &act, NULL);
    pthread_sigmask(SIG_UNBLOCK, &new, NULL);
    printf("\n Press CTRL-C to deliver SIGINT\n");
    sleep(8); /* give user time to hit CTRL-C */
}

void
interrupt(int sig)
{
    printf("thread %d caught signal %d\n", thr_self(), sig);
}
```

- child thread is designated to handle SIGINT, no other thread will get SIGINT
Queueing Abstraction

Ex:
- line at a bank
- multiprocessor executing jobs from a shared job queue
- ER
Arrivals & Departures

- $a_i$: arrival time
- $d_i$: departure time
- $s_i$: service time
- $r_i$: response (system) time
- $q_i$: queueing time

Q1 → S1 → S2

$\lambda$: arrival rate
$\mu$: service rate

$0 \leq t \leq a_i \leq d_i \leq s_i \leq r_i \leq q_i \leq C_1 \leq C_2$
Arrivals & Departures (Cont...)

\[ \lambda \quad Q1 \quad S1 \quad \mu \quad S2 \quad \mu \]

\[ q_1, q_2, q_3 \sim 0 \]
\[ q_4 > 0 \]
Event Driven Simulation

→ An event queue is a sorted list of events according to timestamps; smallest timestamp at the head of queue

→ Object oriented: every object has a "next event" (what it will do next if there is no interference), this event is inserted into the event queue

→ Execution: remove an event from the head of queue, "execute" the event (notify the corresponding object so it can insert the next event)

→ Insert into the event queue according to timestamp of a new event; insertion may cause additional events to be deleted or inserted

→ Potentially repeatable runs (if the same seed is used to initialize random number generator)
Event Driven Simulation (Cont...)

Ex: 4 objects, A (arrival), Q1 (passive object, does not generate events), S1, S2

Initially:
- A : [ a₁, create(C₁) ]
- Q1 : empty
- S1 : NULL
- S2 : NULL

only one event, next event to fire is [ a₁, create(C₁) ]
create(C₁), Q1->enqueue(C₁)
Q1->dequeue(C₁), S1->serve(C₁)

- A : [ a₂, create(C₂) ]
- Q1 : empty
- S1 : [ d₁, destroy(C₁) ]
- S2 : NULL

min(a₂, d₁) = a₂, next event to fire is [ a₂, create(C₂) ]
create(C₂), Q1->enqueue(C₂)
Q1->dequeue(C₂), S2->serve(C₂)

- A : [ a₃, create(C₃) ]
- Q1 : empty
- S1 : [ d₁, destroy(C₁) ]
- S2 : [ d₂, destroy(C₂) ]
Event Driven Simulation (Cont...)

- \( \min(a_3, d_1, d_2) = d_1 \), next event to fire is [ \( d_1 \), destroy(\( C_1 \)) ]
  destroy(\( C_1 \))
  - A : [ \( a_3 \), create(\( C_3 \)) ]
  - S1 : NULL
  - Q1 : empty
  - S2 : [ \( d_2 \), destroy(\( C_2 \)) ]

- \( \min(a_3, d_2) = a_3 \), next event to fire is [ \( a_3 \), create(\( C_3 \)) ]
  create(\( C_3 \)), Q1->enqueue(\( C_3 \))
  Q1->dequeue(\( C_3 \)), S1->serve(\( C_3 \))
  - A : [ \( a_4 \), create(\( C_4 \)) ]
  - S1 : [ \( d_3 \), destroy(\( C_3 \)) ]
  - S2 : [ \( d_2 \), destroy(\( C_2 \)) ]
  - Q1 : empty

- \( \min(a_4, d_2, d_3) = a_4 \), next event to fire is [ \( a_4 \), create(\( C_4 \)) ]
  create(\( C_4 \)), Q1->enqueue(\( C_4 \))
  - A : [ \( a_5 \), create(\( C_5 \)) ]
  - S1 : [ \( d_3 \), destroy(\( C_3 \)) ]
  - S2 : [ \( d_2 \), destroy(\( C_2 \)) ]
  - Q1 : \( C_4 \)

- etc.
Event Driven Simulation (Cont...)

The diagram illustrates the concept of event-driven simulation with time (t) as the horizontal axis and various events labeled as $C_1$, $C_2$, $C_3$, $C_4$, $d_1$, $d_2$, $d_3$, and $d_4$. The events $a_1$, $a_2$, $a_3$, and $a_4$ are shown at different time points, each associated with a specific event. The diagram helps visualize the progress of events over time in an event-driven simulation.
**Time Driven Simulation**

- Every active object is a thread
  - a customer is a passive object, it gets passed around

- To execute a job for $x$ msec, the thread sleeps for $x$ msec
  - nunki.usc.edu does not run a realtime OS
  - it may not get woken up more than $x$ msec later, and sometimes, *a lot more* than $x$ msec later
    - you need to decide if the extra delay is reasonable or it is due to a bug in your code

- Let your machine decide which thread to run next (irreproducible results)

- Compete for resources (such as Q1), must use mutex
You will need to implement 3 threads (or 1 main thread and 3 child threads)

- the *arrival thread* sits in a loop
  - sleeps for an interval, trying to match a given interarrival time (from trace or coin flip)
  - wakes up, creates a customer object, enqueues the customer to Q1, and goes back to sleep
  - if the Q1 was empty before, need to *signal* or *broadcast* a *queue-not-empty condition*

- two *server threads*
  - initially blocked, *waiting* for the *queue-not-empty condition* to be *signaled*
  - (cont...)
Time Driven Simulation (Cont...)

- two server threads (cont...)
  - when it is unblocked, if Q1 is not empty, dequeues a customer, sleeps for an interval matching the service time of the customer, eject the customer from the system, check if Q1 is empty, etc.
  - if there is no work to perform, go wait for the queue-not-empty condition to be signaled

<Ctrl+C>
  - arrival thread will stop generating customers and terminate
    - the arrival thread needs to clear out Q1
  - server threads must finish serving its current customer
  - must print statistics for all customer seen
Time Driven Simulation (Cont...)

Notation:  
- $\alpha_i$ : inter-arrival time for customer $i$ ($a_i - a_{i-1}$), $a_0 = 0$
- $\beta_i$ : service time of customer $i$

- Initially:
  - A : sleep($\alpha_1 = a_1$)
  - Q1 : empty
  - S1 : idle
  - S2 : idle

- A wakes up at $a_1$: create($C_1$), Q1->enqueue($C_1$)
  - Q1->dequeue($C_1$), S1->serve($C_1$)

- A wakes up at $a_1 + \alpha_2$:
  - A : sleep($\alpha_2$)
  - Q1 : empty
  - S1 : sleep($\beta_1$)
  - S2 : idle

- A wakes up at $a_1 + \alpha_2$:
  - A : sleep($\alpha_3$)
  - Q1 : empty
  - S1 : sleeping...
  - S2 : sleep($\beta_2$)

- etc.
Coin Flipping

- Uniform distribution
  - probability mass function (pmf), denoted by \( f(x) \)
    \[
    \int_{-\infty}^{\infty} f(x) \, dx = 1
    \]

- Probability Distribution Function (PDF), denoted by \( F(x) \)
  \[
  F(x) = \int_{-\infty}^{x} f(w) \, dw
  \]

\[
F(x) = \begin{cases} 
0 & x < 0 \\
1 & 0 \leq x \leq 1 \\
1 & x > 1 
\end{cases}
\]
How do you flip a coin according to this distribution?

Think about discrete case:

Add them up:

Flip a coin between 0 and 23

\[ r = \text{drand48}() \times 23 \]

\( r \) lies between 3 and 13, so we have randomly chosen bucket #2
Q: What were we doing when we "added them up"?

A: We were doing "integration".

Hint: $0 \leq F(x) \leq 1$ for any $x$

Can numerically compute $w$
Coin Flipping (Cont...)

Exponential distribution

Note: inter-arrival time of a Poisson process is Exponentially distributed

\[ f(x) = me^{-mx} \]

\[ F(x) = \int_0^x f(y)dy = 1 - e^{-mx} \]

\[ r = \text{drand48}() \]
\[ w = ? \]
Calculating Statistics

arrival thread timeout (read clock)
lock & unlock stdout to print arrival msg
try lock mutex to enter Q
enter Q (read clock)
unlock mutex
lock & unlock stdout to print enter queue msg
try lock mutex to leave Q
leave Q (read clock)
unlock mutex
lock & unlock stdout to print leave queue and begin service msgs
begin service
leave server
lock & unlock stdout to print msg

overhead?
time in Q

time in server

select()?
charge to no one

time

\[ \text{time between begin service and leave server is the amount of time in select(())} \]
Mean and Standard Deviation

- **Average time**
  - for $n$ samples, add up all the time and divide by $n$

- **Average number of customer at a server**
  - same a fraction of time the server is busy

- **Average number of customer at Q1**

- **Standard deviation is the squareroot of variance**
  - $\text{Var}[X] = E[X^2] - (E[X])^2$