P3 - Jurassic Park

Motivation

Contemporary operating systems are built around the concept of processes or tasks. Tasks may or may not be related, may or may not affect the state of another task, but usually always need to share resources in a protected, prioritized, and equitable manner.

Jurassic Park is an inter-process communication and synchronization problem between multiple tasks. Visitors to the park want to first get a ticket, visit the park museum, take a ride on a park tour car to see the dinosaurs, and then visit the gift shop before leaving the park. Since the park is on a limited budget, drivers must perform double duty when not sleeping; selling tickets to the park visitors and driving the tour cars through the park. Visitors, drivers, and cars are represents by concurrent tasks while an additional task is used to display the park status every second.

Attempting to coordinate the park activities without bringing about any race conditions is typical of many queuing problems. A poorly implemented solution could lead to the inter-process communication problems of starvation and deadlock. For example, the driver could end up waiting to sell a ticket to a visitor while a loaded tour car could be waiting for a driver, resulting in deadlock. Alternatively, visitors may not decide to take a tour car ride in an orderly manner, leading to process starvation as some visitors never get the chance for a ride even though they have been waiting in line.

Jurassic Park is not unlike real world simulation and control problems such as found in airports, supply houses, traffic systems, oil tank farms, etc. Each requires multitasking roles involving inter-process coordination and communication. Like the classical dining philosophers and sleeping barber problems, the Jurassic Park exercise demonstrates what problems can occur in a multitasking, multi-user environment when many tasks are competing for the same resources.

Project Description

Visitors arrive at the Jurassic Park at random times. (Only 20 visitors may be in the park at any one time - OSHA requirements!) Upon entering the park, a visitor gets in line to purchase a ticket. After obtaining a ticket from a driver, the visitor gets in the museum line. (Again, a limited number of visitors are allowed in the museum as well as the gift shop.) After visiting the museum, the visitor gets in the tour car line to wait until invited to board a tour car. As a visitor boards a tour car, he returns his ticket. When the touring car is filled with visitors (3) and a driver, the car enters Jurassic Park and follows a guided tour path through the park. When the tour car completes the ride and pulls into the unloading station, the visitors debark and move to the gift shop line, the driver goes back to sleep awaiting new duties, and the tour car pulls forward to be loaded again. After visiting the gift shop, the visitor exits the park.

Project Requirements

The following are important guidelines for completing the Jurassic Park assignment:

1. Forty-five visitors must pass successfully thru the park.
2. Add a delta clock to your operating system. The delta clock ticks in tenth-of-a-second increments. (See Delta Clock.)
3. Create a task for each park visitor (**NUM_VISITORS**), driver (**NUM_DRIVERS**), and tour car (**NUM_CARS**). These tasks should all run at the same priority level.
4. Update the park data structure variables appropriately as visitor, driver, and car states change. The park is displayed using the park data **struct** every second by the jurassicTask task.
5. Each task (visitor, driver, and car) creates its own timing semaphore, which is used for timing functions (ie, arrival delay, standing in lines, time in gift shop or museum, and ride done signals.) The delta clock is used to semSignal these semaphores.
6. Park visitors randomly arrive at the park over a 10 second period. In addition, visitors should stand in lines for a random time before requesting a ticket, entrance to the museum or gift shop, or getting into a tour car (3 seconds maximum).
7. Use resource semaphores (counting) to control access to the park, the number of tickets available, and the number of people allowed in the gift shop and museum.
8. Use mutex semaphores (binary) to protect any critical sections of code within your implementation, such as when updating the delta clock, acquiring a driver to buy a ticket or drive a tour car, accessing global data, or sampling the state of a semaphore.
9. Use semaphores (binary) to synchronize and communicate events between tasks, such as to awaken a driver, signal data is valid, signal a mode change, etc.
10. Use at least one **semTryLock** function in your simulation.
11. The "**SWAP**" directive should be inserted between every line of executable code in your Jurassic Park simulation. Park critical code must be protected by the parkMutex mutex.

12. The park simulation also creates a "**lostVisitor**" task which sums critical variables in the park to detect any lost visitors. Beware!
13. Since all project 3 tasks need to be on the same priority level, it is necessary to let the park fully initialize before proceeding to create your visitor, car, and driver tasks. To that end, execute a "while (!parkMutex) SWAP;" loop just after the **jurassicPark** task is created.
14. You are to implement a fair algorithm that prevents deadlock and starvation rather than detect them.

Delta Clock

A delta clock is a mechanism used to improve performance and simplify the management of timed semaphores. If a linked list (or the equivalent) is used to store pending timed semaphores, then all elements of the list have to be checked every time there is a clock to see if that semaphore is to be signaled. For a small number of semaphores, this might be OK, but certainly not a very scalable solution.

A delta clock structure stores pending timed events such that only the first entry needs to be examined (and updated) when a clock occurs. All other timing events are a function or delta from that first entry.

For example, consider the following timed events:

> After 20 clocks, signal **Event1**
> After 5 clocks, signal **Event2**
> After 35 clocks, signal **Event3**
> After 27 clocks, signal **Event4** and **Event5**
> After 22 clocks, signal **Event6**.

One way to handle this is to create a linked list of events and run through the list on every clock. However, notice that after 5 clocks, the first event to occur is **Event2**. **Event1** occurs 15 clocks later, **Event6** comes 2 clocks after that, **Event4** and **Event5** occur 5 clocks after **Event6**, and finally, **Event3** happens 8 clocks after that. If we capture this information in a delta clock structure, then only the top entry need be decremented on each clock interval. When it tics down to zero, the event is signaled and the delta stack is popped.

The delta clock structure is illustrated below:

Inter-process Communication

Inter-Process Communication (IPC) refers to the exchange of data among two or more threads in one or more processes. Among others, IPC techniques involve synchronization (events and semaphores), shared memory (globals), message passing (pipes and message queues), sockets, and remote procedure calls (RPC). The method of IPC used may vary based on the bandwidth and latency of communication between the threads, and the type of data being communicated. For Jurassic Park, use semaphores and shared memory as follows:

Semaphores are considered events and are usually one of the following types:

1. Resource semaphores. Typically counting, these semaphores are used to control access to the park, the number of tickets available, and the number of people allowed in the gift shop and museum.

```
// create MAX_TICKETS tickets using counting semaphore
tickets = createSemaphore("tickets", COUNTING, MAX_TICKETS); SWAP;
.
.
.
// wait for a ticket (consume)
SEM_WAIT(tickets); SWAP;
.
.
.
// return a ticket (produce)
SEM_SIGNAL(tickets); SWAP;
```

2. Mutex semaphores. Typically binary, these semaphores are used to protect critical sections of code within your implementation, such as when updating the delta clock, acquiring a driver to buy a ticket or drive a tour car, accessing global data, or sampling the state of a semaphore.

```
SEM_WAIT(needDriverMutex); SWAP; // need ticket, wait for driver (mutex)
{
.
.
.
// signal need ticket (signal, put hand up)
.
.
.
}
SEM_SIGNAL(needDriverMutex); SWAP; // release driver (mutex)
```

3. Signal semaphores. Typically binary, these semaphores are used to synchronize and communicate events between tasks, such as to awaken a driver, signal data is valid, signal a mode change, etc. For example, the following code signals the need for a ticket (needTicket), signals a driver to wakeup (wakeupDriver), waits for a ticket to be printed (ticketReady), and then signals to buy the ticket (buyTicket). The task indicates it no longer needs a ticket by consuming the needTicket semaphore.

```
// only 1 visitor at a time requests a ticket
SEM_WAIT(getTicketMutex); SWAP;
{
        // signal need ticket (produce, put hand up)
        SEM_SIGNAL(needTicket); SWAP;

        // wakeup a driver (produce)
        SEM_SIGNAL(wakeupDriver); SWAP;

        // wait for driver to obtain ticket (consume)
        SEM_WAIT(ticketReady); SWAP;

        // put hand down (consume, driver awake, ticket ready)
        // can also be done in the driver task with a sem try wait
        SEM_WAIT(needTicket); SWAP;

        // buy ticket (produce, signal driver ticket taken)
        SEM_SIGNAL(buyTicket); SWAP;
}
```

```
// done (produce)
SEM_SIGNAL(getTicketMutex); SWAP;
```

4. Shared memory can be implemented using C global memory when protected with mutex semaphores.

```
// protect shared memory access
SEM_WAIT(parkMutex); SWAP;

// access inside park variables
myPark.numOutsidePark--; SWAP;
myPark.numInPark++; SWAP;

// release protect shared memory access
SEM_SIGNAL(parkMutex); SWAP;
```

Interfacing with "jurassicTask"

The **jurassicTask** task fills tour cars with passengers when a car is in the loading position, moves tour cars one position forward (if possible) around the park, randomly chooses a route at the route cross roads, and displays the status of Jurassic Park every second. The display is generated from the following C struct variables defined in the os345park.h file:

```
typedef struct
{
        int numOutsidePark;       // # outside of park
        int numInPark;            // # in park (P=#)
        int numTicketsAvailable;  // # left to sell (T=#)
        int numRidesTaken;        // # of tour rides taken (S=#)
        int numExitedPark;        // # who have exited the park
        int numInTicketLine;      // # in ticket line
        int numInMuseumLine;      // # in museum line
        int numInMuseum;          // # in museum
        int numInCarLine;         // # in tour car line
        int numInCars;            // # in tour cars
        int numInGiftLine;        // # in gift shop line
        int numInGiftShop;        // # in gift shop
        int drivers[NUM_DRIVERS]; // driver state (-1=T, 0=z, 1=A, 2=B,
etc.)
        CAR cars[NUM_CARS];       // cars in park
} JPARK;
```

Your visitor, driver, and tour car tasks need to update the appropriate variables at the appropriate times. The following global signal semaphores are defined in "os345park.c":

```
Semaphore* fillSeat[NUM_CARS];
Semaphore* seatFilled[NUM_CARS];
Semaphore* rideOver[NUM_CARS];
```

All car tasks should

1. for each car (and seat in the car),
    - wait (fillSeat) to be filled, seat by seat, when the car is in loading position,
    - get a passenger from the tour car waiting line,
    - get a driver for car (if last seat)
    - signal (seatFilled) that seat has been filled,
2. wait for a signal (rideOver) that the ride is over,
3. and release passenger and driver (allow driver to go back to sleep.)

For example, to fill a seat in tour car carID, wait for available seat ready (fillSeat[carID]), then find a visitor from the tour car waiting line, and then signal (seatFilled[carID]) that you are ready for the next seat. (Communication with the park is indicated in RED.)

```
SEM_WAIT(fillSeat[carID]);              SWAP    // wait for available seat

SEM_SIGNAL(getPassenger);               SWAP;   // signal for visitor
SEM_WAIT(seatTaken);                    SWAP    // wait for visitor to reply

... save passenger ride over semaphore ...

SEM_SIGNAL(passengerSeated);    SWAP:   // signal visitor in seat

// if last passenger, get driver
{
        SEM_WAIT(needDriverMutex);      SWAP;

          // wakeup attendant
          SEM_SIGNAL(wakeupDriver);     SWAP;

        ... save driver ride over semaphore ...

          // got driver (mutex)
          SEM_SIGNAL(needDriverMutex);  SWAP;
}

SEM_SIGNAL(seatFilled[carID]);  SWAP ;  // signal ready for next seat
```

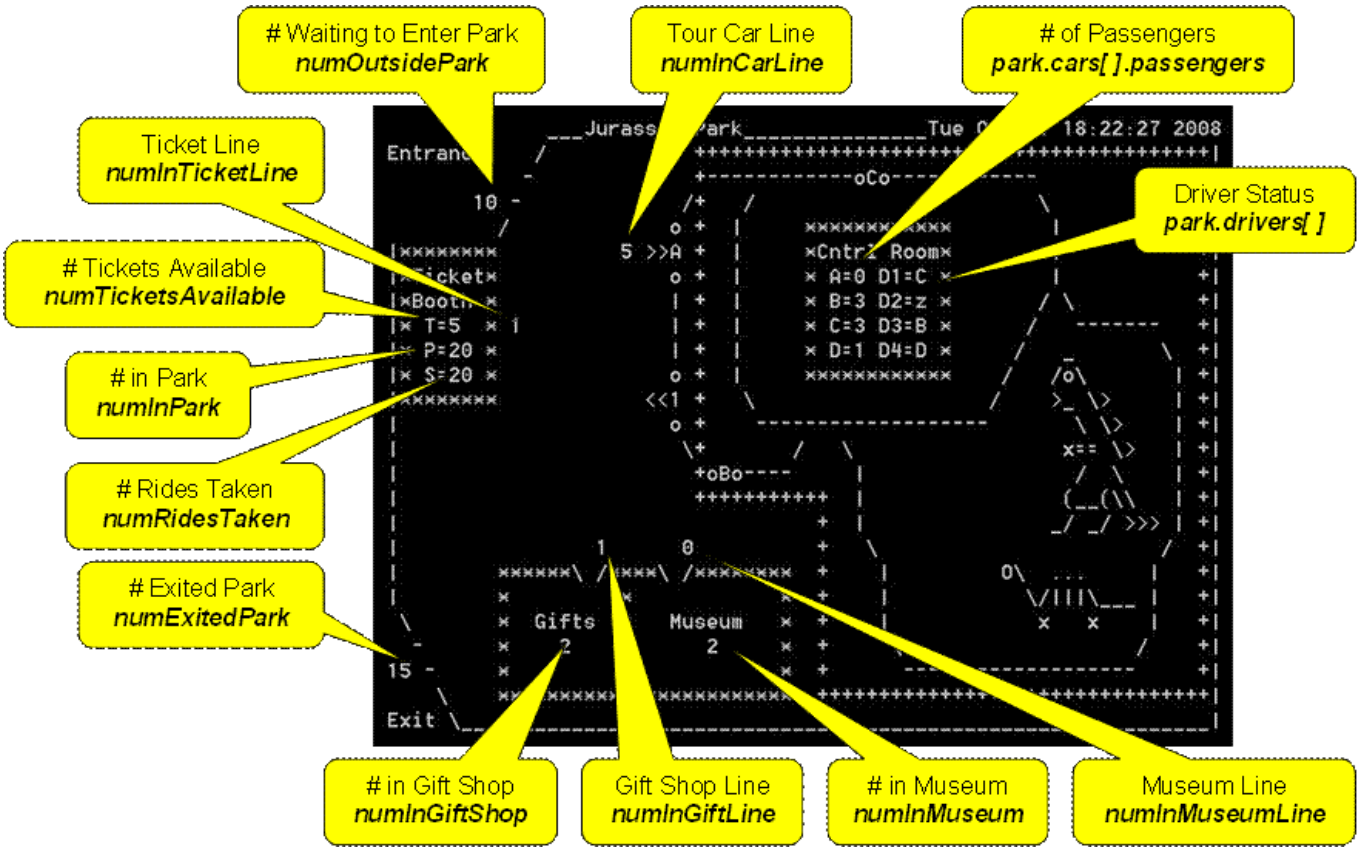
```
// if car full, wait until ride over
SEM_WAIT(rideOver[myID]);                SWAP

... release passengers and driver ...
```

In summary:

Action	Car Task		Park Task
For each car seat:	SEM_WAIT(fillSeat[carID]); Get passenger Save passenger rideDone[] semaphore Get driver (if last passenger) Save driver driverDone semaphore SEM_SIGNAL(seatFilled[carID]);	← →	SEM_SIGNAL(fillSeat[carID]); SEM_WAIT(seatFilled[carID]);
Wait until ride over	SEM_WAIT(rideOver[carID]);	←	SEM_SIGNAL(rideOver[carID]);
Release driver	SEM_SIGNAL(driverDone);		
Release passengers	SEM_SIGNAL(rideDone[i]);		

Park variables correspond to the park display as follows:



Suggested Steps to Implementing the Jurassic Park Project

1. Implement delta clock.
   - Design data structure to hold delta times/events.

- Add 1/10 second routine to pollinterrupts. Decrement top event and semSignal when time = 0.
- Program an insert delta clock routine (insertDeltaClock(int time, Semaphore* sem)).
- Thoroughly test the operation of your delta clock before proceeding.

2. Develop the car task.
    - Design car functionality and Jurassic Park interface. (Don't worry about passengers yet.)
    - Implement design and integrate with os345 and Jurassic Park.
    - Validate correct car behaviour.
3. Develop the visitor task.
    - Design visitor functionality and car task interface.
    - Implement design and integrate with os345 and car tasks. (Don't worry about tickets yet.)
    - Use delta-clock to vary visitor time in all lines, museum, and gift shop.
    - Observe correct visitor behavior as a visitor moves through the park.
4. Develop the driver task.
    - Design driver functionality and interface with visitor and car tasks.
    - Implement design and integrate with os345, visitor, and car tasks. (Now is the time to worry about ticket sales and driver duties.)
    - Add ticket sales and driver responsibilities.
    - When a driver is awakened, use the semTryLock function to determine if a driver or a ticket seller is needed.

Grading and Pass-off

Your Jurassic Park Lab is to be demonstrated in person with a TA. The assignment is worth 10 points, which will be awarded as follows:

points	*requirments*
3	Implement a delta clock in the pollInterrupts routine (OS345.c) to signal semaphores after a certain time period, unblocking sleeping tasks. (The delta clock should tick down every 1/10 of a second.) Design the delta clock to signal any type of semaphore, binary or counting. The clock should handle mutual exclusion with insert/delete routines and not lose any time. Implement functions to insert and delete semaphores from the delta clock. These routines must properly handle mutual exclusion.

---

points	*requirments*
2	Create a single, re-entrant visitor task that 1. tries to enter the Jurassic Park at random times over a 10 second period. (Use a counting semaphore to restrict the number of visitors in the park to MAX_IN_PARK.) 2. upon being allowed in the park, a visitor must get in line to purchase a ticket. (Use a counting semaphore to restrict the number of available tickets to MAX_TICKETS.) 3. after successfully obtaining a ticket from a free driver, the visitor gets in the museum line. (Again, use counting semaphores to restrict the number of visitors in the museum and gift shop to MAX_IN_MUSEUM and MAX_IN_GIFTSHOP respectively.) 4. after visiting the museum, the visitor gets in the tour car line to wait until permitted to board a tour car. (Release visitor ticket after boarding tour car.) 5. when the touring car ride is over, the visitor moves to the gift shop line. 6. after visiting the gift shop, the visitor exits the park allowing another visitor into the park.
1	Create a single, re-entrant car task, which acquires (NUM_SEATS) visitors as it waits (fillSeat) and fills (seatFilled) requested car seats, acquires a driver, waits (rideOver) until the park ride is over, and then releases the driver/visitors as described above.
1	Create a single, re-entrant driver task, which waits to be awakened and then either sells a visitor a ticket or else fills a driver seat in a car and waits until the ride is over. Use the semTryLock function to acquire the correct resource semaphore. (The driver task controls ticket access using a resource semaphore.)
1	Use resource semaphores (counting) to control access to the park, the number of tickets available, and the number of people allowed in the gift shop and museum. Use mutex semaphores (binary) to protect any critical sections of code within your implementation, such as when updating the delta clock, acquiring a driver to buy a ticket or drive a tour car, accessing global data, or sampling the state of a semaphore. Use signal semaphores (binary) to synchronize and communicate events between tasks, such as to awaken a driver, signal data is valid, signal a mode change, etc.

points	*requirments*
2	Have the project3 command schedule the jurassicTask. Also schedule 4 driver tasks, 4 car tasks, and 45 visitor tasks. All Jurassic Park tasks should execute at the same priority level. Observe proper behavior as all visitors visit the museum, take a tour car ride, visit the gift shop, and exit the park. Make sure that a SWAP directive is placed after every C instruction in your Jurassic Park visitor, car, and driver simulation code (not kernel routines) as well as any non-kernel Delta Clock code. These context switches will verify that mutual exclusion is properly implemented for a truly pre-emptive environment.

In addition, **after completing the above requirements**, the following bonus points may be awarded: