HW1
and
Synchronization & Queuing

CS241 Discussion Section
Spring 2009
Week 7

TA Review

- Optional TA Review Session

The TAs will hold an optional review session to answer last-minute exam questions:

Saturday, March 14, 2009
2pm – 4pm
2405 SC

Outline

- HW1 Discussion
- Synchronization problems
  - Producer-consumer
  - Dining Philosophers
- Queueing theory

HW1

a) int *x, q=0;
*x = q; (a): _____

b) int b[2];
*b = 20; (b): _____

c) char greet[80];
strcpy (‘‘Hello’’, greet); (c): _____

HW1

a) int *x, q=0;
*x = q; (a): Incorrect

b) int b[2];
*b = 20; (b): Correct

c) char greet[80];
strcpy (‘‘Hello’’, greet); (c): Incorrect

HW1

d) int *p, q[2];
p = malloc (sizeof (int));
*p = 3;
q[2]=*p; (d): _____

e) int *x, y;
x = &y;
*x = 10; (e): _____

HW1

d) int *p, q[2];
p = malloc (sizeof (int));
*p = 3;
q[2]=*p; (d): Incorrect

e) int *x, y;
x = &y;
*x = 10; (e): Correct

HW1

Which scheduling policy

a) minimizes average task waiting time? _____

b) minimizes average task response time? _____

c) suffers the convoy effect? _____

d) has the largest context switch overhead? _____

e) may suffer a starvation effect? _____

HW1

Which scheduling policy

a) minimizes average task waiting time? ___SJF_____

b) minimizes average task response time? ____RR_____

c) suffers the convoy effect? ____FIFO_____

d) has the largest context switch overhead? ____RR_____

e) may suffer a starvation effect? ___SJF___

HW1

```
while (x > 0) {};
        x ++; /* assume this line is atomic */
        execute critical section;
        x – –; /* assume this line is atomic */
```

Mutual exclusion:
Progress:

HW1

```
while (x > 0) {};
        x ++; /* assume this line is atomic */
        execute critical section;
        x – –; /* assume this line is atomic */
```

Mutual exclusion: No
Progress: Yes

HW1

x2 = 1;	x1 = 1;
while (x1 != 0) {};	while (x2 != 0) {};
critical section;	critical section;
x2 = 0;	x1 = 0;

Mutual exclusion:
Progress:

HW1

```
x2 = 1;                    x1 = 1;
while (x1 != 0) {};        while (x2 != 0) {};
critical section;          critical section;
x2 = 0;                    x1 = 0;
```

Mutual exclusion: Yes
Progress: No

HW1

```
while (y = = 1) {};        while (y = = 0) {};
y = 1;                      y = 0;
critical section;          critical section;
```

Mutual exclusion:
Progress:

HW1

```
while (y = = 1) {};        while (y = = 0) {};
y = 1;                      y = 0;
critical section;          critical section;
```

Mutual exclusion: No
Progress: No

HW 1

a) Function g() calls function f().
b) Function g() creates a POSIX thread that executes function f().

a) Kill –
b) Exec –
c) Exit –

HW 1

a) Function g() calls function f().
control passes from function g() to function f() then returns to g() when f() ends.

b) Function g() creates a POSIX thread that executes function f().
when the new thread is created, f() executes concurrently with g().

a) Kill – sends a signal to a process
b) Exec – runs a new program in the current address space
c) Exit – terminates a unix process

HW1

- Change behavior of SIGUSR1
- Block SIGINT

Which of these?

sleep alarm sigwait
sigprocmask sigaction sigsuspend

HW1

- Change behavior of SIGUSR1
- Block SIGINT

Which of these?

sleep alarm sigwait
sigprocmask sigaction sigsuspend

HW1

a) The turnaround time minus the _____ time is equal to the waiting time.

b) The turnaround time is the time from the process creation time to its _____ time.

b) The total time a process spends in queues is called _____ time

d) Response time is the interval from _____ time to _____ time

HW1

a) The turnaround time minus the execution time is equal to the waiting time.

b) The turnaround time is the time from the process creation time to its finish time.

b) The total time a process spends in queues is called waiting time

d) Response time is the interval from arrival time to start time

Example 1: Producer-Consumer Problem

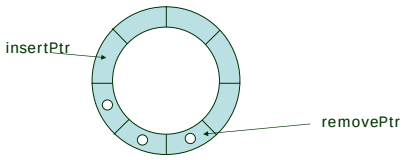
Producers insert items

Consumers remove items

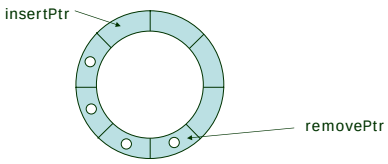
Shared bounded buffer

e.g. a circular buffer with an insert and a removal pointer.

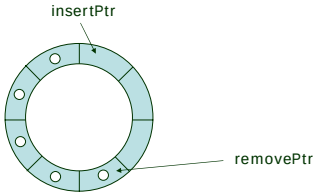
Producer-Consumer



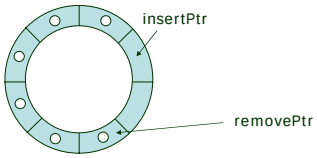
Producer-Consumer



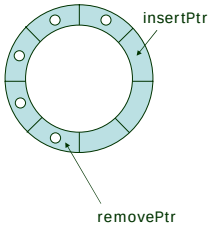
Producer-Consumer



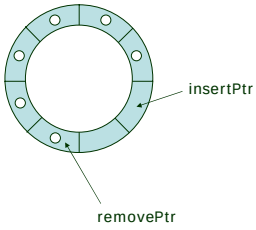
Producer-Consumer



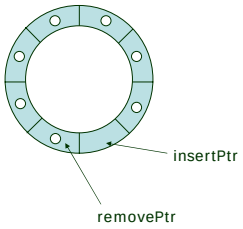
Producer-Consumer



Producer-Consumer

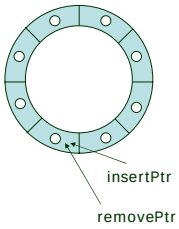


Producer-Consumer

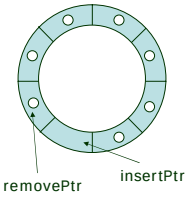


Producer-Consumer

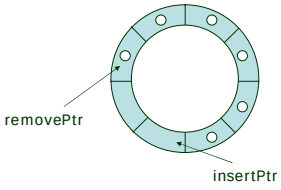
BUFFER FULL: Producer must be blocked!



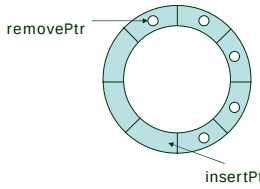
Producer-Consumer



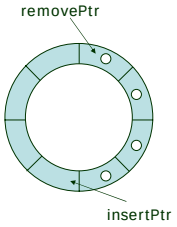
Producer-Consumer



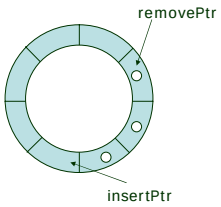
Producer-Consumer



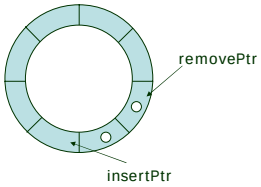
Producer-Consumer



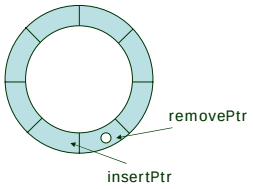
Producer-Consumer



Producer-Consumer

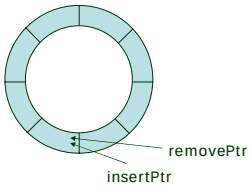


Producer-Consumer



Producer-Consumer

BUFFER EMPTY: Consumer must be blocked!



Challenge

Need to prevent:

Buffer Overflow
- Producer writing when there is no storage

Buffer Underflow
- Consumer reading nonexistent data

Race condition
- Two processes editing the list at the same time

Synchronization variables

Create these variables to prevent these problems:

`items` semaphore
- Counts how many items are in the buffer
- Cannot drop below 0

`slots` semaphore
- Counts how may slots are available in the buffer
- Cannot drop below 0

`list` mutex
- Makes buffer access mutually exclusive

Producer-Consumer Example

`ds7-problem1.c` shows an example implementation for one producer and one consumer, but without synchronization code.

- Running it shows
  - Buffer underflows
    - Nonsense data is consumed
  - Buffer overflows
    - Unconsumed data is overwritten

Example 2: Dining Philosophers

Dining Philosopher Challenge

{ Think | Eat }

N Philosophers circular table with *N* chopsticks

To eat the Philosopher must first pickup two chopsticks

i^{th} Philosopher needs i^{th} & $i+1^{st}$ chopstick

Only put down chopstick when Philosopher has finished eating

Devise a solution which satisfies mutual exclusion but avoids starvation and deadlock

The simple implementation

```
while(true) {
   think()
   lock(chopstick[i])
   lock(chopstick[(i+1) % N])
   eat()
   unlock(chopstick[(i+1) % N])
   unlock(chopstick[i])
}
```

Does this work?

Deadlocked!

When every philosopher has picked up his left chopstick, and no philosopher has yet picked up his right chopstick, no philosopher can continue.

Each philosopher waits for his right neighbor to put a chopstick down, which he will never do.

This is a *deadlock*.

Formal Requirements for Deadlock

Mutual exclusion
- Exclusive use of chopsticks

Hold and wait condition
- Hold 1 chopstick, wait for next

No preemption condition
- Cannot force another to undo their hold

Circular wait condition
- Each waits for next neighbor to put down chopstick

The simple implementations satisfies all of these.

Problems for Week 7

1) `ds7-problem1.c` contains producer-consumer code.

Use the provided semaphores and mutexes to prevent buffer overflows, underflows, and race conditions.

Think: When should the consumer block? When should the producer block?

Problems for Week 7 (contd)

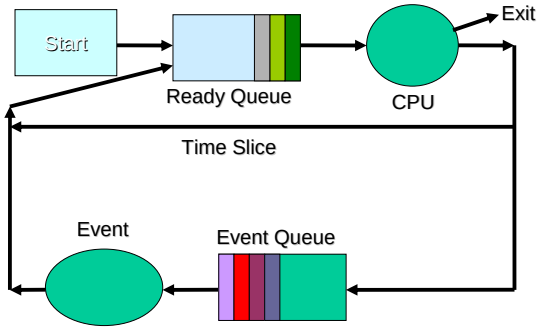
2) `ds7-problem2.c` contains dining philosophers code.

Alter the program to prevent deadlock. There are multiple ways to do this.

Think: What are the conditions of deadlock? Can any of them be removed?

Queuing Theory

Queuing Diagram for Processes

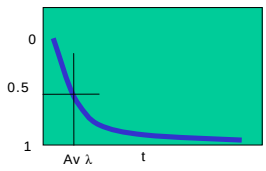


Queueing Theory

Steady state

Poisson arrival with λ constant arrival rate (customers per unit time) each arrival is independent.

$P(\tau \leq t) = 1 - e^{-\lambda t}$



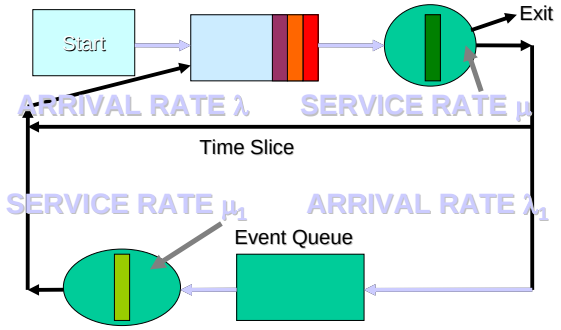
Analysis of Queueing Behavior

Probability *n* customers arrive in time interval *t* is:

$$e^{-\lambda t}(\lambda t)^{n}/\, n!$$

Assume random service times (also Poisson): μ constant service rate (customers per unit time)

Queuing Diagram for Processes



Useful Facts From Queuing Theory

W_q = mean time a customer spends in the queue
λ = arrival rate
$L_q = \lambda W_q$ number of customers in queue
W = mean time a customer spends in the system
$L = \lambda W$ (Little's theorem) number of customers in the system

In words – average length of queue is arrival rate times average waiting time

Analysis of Single Server Queue

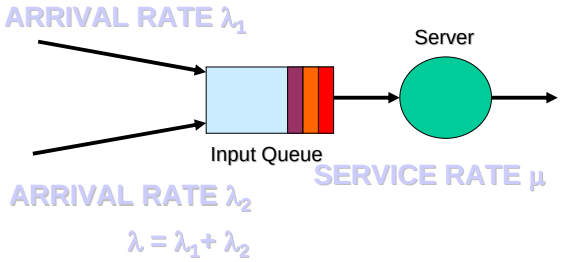
Server Utilization: $\rho = \lambda/\mu$

Time in System: $W = 1/(\mu-\lambda)$

Time in Queue: $W_q = \rho/(\mu-\lambda)$

Number in Queue (Little): $L_q = \rho^2/(1-\rho)$

Poisson Arrival & Service Rates Sum



TA Review

- Optional TA Review Session

  The TAs will hold an optional review session to answer last-minute exam questions:

  Saturday, March 14, 2009
  2pm – 4pm
  2405 SC