Operating Systems

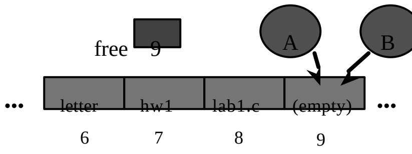
Process Synchronization

Too Much Pizza

	Person A	Person B
3:00	Look in fridge. Pizza!	
3:05	Leave for store.	Look in fridge. Pizza!
3:10	Arrive at store.	Leave for store.
3:15	Buy pizza.	Arrive at store.
3:20	Arrive home.	Buy pizza.
3:25	Put away pizza.	Arrive home.
3:30		Put pizza away. Oh no!

Cooperating Processes

- Consider: print spooler
  - Enter file name in spooler queue
  - Printer daemon checks queue and prints



- "Race conditions" (ugh!)
- (Hey, you! Show demo!)

Producer Consumer

- Model for cooperating processes
- Producer "produces" and item that consumer "consumes"
- Bounded buffer (shared memory)

```
item buffer[MAX]; /* queue */
int counter; /* num items */
```

Producer

```
item i; /* item produced */
int in; /* put next item */
while (1) {
  produce an item
  while (counter == MAX){/*no-op*/}
  buffer[in] = item;
  in = (in + 1) % MAX;
  counter = counter + 1;
}
```

Consumer

```
item i; /* item consumed */
int out; /* take next item */
while (1) {
  while (counter == 0) {/*no-op*/}
  item = buffer[out];
  out = (out + 1) % MAX;
  counter = counter - 1;
  consume the item
}
```

Trouble!

```
P:   R1 = counter     {R1 = 5}
P:   R1 = R1 + 1      {R1 = 6}
C:   R2 = counter     {R2 = 5}
C:   R2 = R2 -1       {R2 = 4}
C:   counter = R2     {counter = 4}
P:   counter = R1     {counter = 6}
```

Critical Section

- ✦ Mutual Exclusion
  - – Only one process inside critical region
- ✦ Progress
  - – No process outside critical region may block other processes wanting in
- ✦ Bounded Waiting
  - – No process should have to wait forever (starvation)
- ✦ Note, no assumptions about speed!

First Try: Strict Alternation

```
int turn; /* shared, i or j */

while(1) {
  while (turn <> i) { /* no-op */}
  /* critical section */
  turn = j
  /* remainder section */
}
```

Second Try

```
int flag[1]; /* boolean */

while(1) {
  flag[i] = true;
  while (flag[j]) { /* no-op */}
  /* critical section */
  flag[i] = false;
  /* remainder section */
}
```

Third Try: Peterson’s Solution

```
int flag[1]; /* boolean */
int turn;
while(1) {
  flag[i] = true;
  turn = j;
  while (flag[j] && turn==j){ }
  /* critical section */
  flag[i] = false;
  /* remainder section */
}
```

Multiple-Processes

- ✦ “Bakery Algorithm”
- ✦ Common data structures

  `boolean choosing[n];`

  `int num[n];`
- ✦ Ordering of processes
  - – If same number, can decide “winner”

Multiple-Processes

```
choosing[i] = true;
num[i] = max(num[0],num[1] …)+1
choosing[i] = false;
for (j=0; j<n; j++) {
 while(choosing[j]) { }
 while( num[j]!=0 &&
   (num[j],j)<(num[i],i) ) {}
}
/* critical section */
num[i] = 0;
```

Synchronization Hardware

- Test-and-Set: returns and modifies atomically

```
int Test_and_Set(int target) {
 int temp;
 temp = target;
 target = true;
 return temp;
}
```

Synchronization Hardware

```
while(1) {
 while (Test_and_Set(lock)) { }
 /* critical section */
 lock = false;
 /* remainder section */
}
```

Questions

- What is a “race condition”?
- What are 3 properties necessary for a correct “critical region” solution?
- What is one drawback of both Peterson’s solution and Test_and_Set hardware?

Semaphores

- Does not require “busy waiting”
- Semaphore S (shared, often initially =1)
  - integer variable
  - accessed via two (indivisible) atomic operations

```
wait(S): S = S - 1
  if S<0 then block(S)
signal(S): S = S + 1
  if S<=0 then wakeup(S)
```

Critical Section w/Semaphores

```
semaphore mutex; /* shared */

while(1) {
 wait(mutex);
 /* critical section */
 signal(mutex);
 /* remainder section */
}
```

(Hey, you! Show demo!)

Semaphore Implementation

- How do you make sure the *signal* and the *wait* operations are atomic?

Semaphore Implementation

- Disable interrupts
  - Why is this not evil?
  - Multi-processors?
- Use correct software solution
- Use special hardware, i.e.- Test-and-Set

Design Technique: Reducing a Problem to a Special Case

- Simple solution not adequate
  - ex: disabling interrupts
- Problem solution requires special case solution
  - ex: protecting S for semaphores
- Simple solution adequate for special case
- Other examples:
  - name servers, on-line help

Trouble!

```
signal(S)
/* cr */
wait(S)
```

```
wait(S)
/* cr */
wait(S)
```

```
/* cr */
```

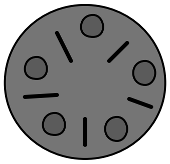
Process A	*Process B*
`wait(S)`	`wait(Q)`
`wait(Q)`	`wait(S)`
...	...

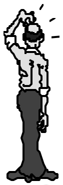
Classical Synchronization Problems

- Bounded Buffer
- Readers Writers
- Dining Philosophers

Dining Philosophers

- Phisolophers
  - Think
  - Sit
  - Eat
  - Think
- Need 2 chopsticks to eat





Dining Philosophers

<u>Philosopher i:</u>

```
while (1) {
  /* think… */
  wait(chopstick[i]);
  wait(chopstick[i+1 % 5]);
  /* eat */
  signal(chopstick[i]);
  signal(chopstick[i+1 % 5]
}
```

Other Solutions?

Other Solutions

- ✦ Allow at most N-1 to sit at a time
- ✦ Allow to pick up chopsticks only if both are available
- ✦ Asymmetric solution (odd L-R, even R-L)

Outline

- ✦ Need for synchronization
  - – why?
- ✦ Solutions that require busy waiting
  - – what?
- ✦ Semaphores
  - – what are they?
- ✦ Classical problems
  - – dining philosophers
  - – reader/writers (today)

Readers-Writers

- ✦ *Readers* only read the content of object
- ✦ *Writers* read and write the object
- ✦ Critical region:
  - – No processes
  - – One or more readers (no writers)
  - – One writer (nothing else)
- ✦ Solutions favor Reader *or* Writer

Readers-Writers

<u>Shared:</u>

```
semaphore mutex, wrt;
int readcount;
```

<u>Writer:</u>

```
wait(wrt)
/* write stuff */
signal(wrt);
```

Readers-Writers

<u>Reader:</u>

```
wait(mutex);
readcount = readcount + 1;
if (readcount==1) wait(wrt);
signal(mutex);
/* read stuff */
wait(mutex);
readcount = readcount - 1;
if (readcount==0) signal(wrt);
signal(mutex);
```

Monitors

- High-level construct
- Collection of:
  - variables
  - data structures
  - functions
  - Like C++ class
- One process active inside
- “Condition” variable
  - not counters like semaphores

Monitor Producer-Consumer

```
monitor ProducerConsumer {
   condition full, empty;
   integer count;

   /* function prototypes */
   void enter(item i);
   item remove();
}
void producer();
void consumer();
```

Monitor Producer-Consumer

```
void producer() {
  item i;
  while (1) {
     /* produce item i */
     ProducerConsumer.enter(i);
  }
}
void consumer() {
  item i;
  while (1) {
     i = ProducerConsumer.remove();
     /* consume item i */
  }
```

Monitor Producer-Consumer

```
void enter (item i) {
   if (count == N) wait(full);
   /* add item i */
   count = count + 1;
   if (count == 1) then signal(empty);
}
item remove () {
  if (count == 0) then wait(empty);
  /* remove item into i */
  count = count - 1;
  if (count == N-1) then signal(full);
  return i;
}
```

Other IPC Synchronization

- Critical Regions
- Conditional Critical Regions
- Sequencers
- Path Expressions
- Serializers
- ...
- All essentially equivalent in terms of semantics. Can build each other!

Ex: Cond. Crit. Region w/Sem

```
region X when B do S {
 wait(x-mutex);
 if (!B) {
   x-count = x-count + 1;
   signal(x-mutex);
   wait(x-delay);
   /* wakeup loop */
   x-count = x-count -1
}
/* remainder */
```

Ex: Wakeup Loop

```
while (!B) {
  x-temp = x-temp + 1;
  if (x-temp < x-count)
    signal(x-delay);
  else
    signal(x-mutex);
  wait(x-delay);
}
```

Ex: Remainder

```
S;
if (x-count > 0) {
  x-temp = 0;
  signal(x-delay);
} else
  signal(x-mutex);
```

Trouble?

- Monitors and Regions attractive, but ...
  - Not supported by C, C++, Pascal ...
    - semaphores easy to add
- Monitors, Semaphores, Regions ...
  - require shared memory
  - break on multiple CPU (w/own mem)
  - break distributed systems
- Message Passing!

Message Passing

- Communicate information from one process to another via primitives:
  `send(dest, &message)`
  `receive(source, &message)`
- Receiver can specify *ANY*
- Receiver can block (or not)

Producer-Consumer

```
void Producer() {
   while (TRUE) {
      /* produce item */
      build_message(&m, item);
      send(consumer, &m);
      receive(consumer, &m); /* wait for ack */
   }}
void Consumer {
  while(1) {
      receive(producer, &m);  [redacted]
      extract_item(&m, &item);
      send(producer, &m); /* ack */
      /* consume item */
  }}
```

Consumer Mailbox

```
void Consumer {
  for (i=0; i<N; i++)
     send(producer, &m);  /* N empties */
  while(1) {
     receive(producer, &m);
     extract_item(&m, &item);
     send(producer, &m);  /* ack */
     /* consume item */
  }
}
```

New Troubles with Messages?

New Troubles

- Scrambled messages (*checksum*)
- Lost messages (*acknowledgements*)
- Lost acknowledgements (*sequence no*.)
- Process unreachable (down, terminates)
- Naming
- Authentication
- Performance (from copying, message building)
- (Take cs4514!)