CSc33200: Operating Systems, CS-CCNY, Fall 2003

Midterm Exam Solutions

November 23, 2003

The highest score is 175 and the average is around 115. Since the scores are significantly lower than previous homework and projects, everyone will receive an increase of 30 points.

1. The Pumpkin Computer uses a segmented addressing scheme in which individual bytes are accessed by combining a 16-bit segment paragraph and a 16-bit relative offset. SR is 16-bit register that points to the beginning of a 16-byte paragraph that is evenly divisible by 16. The segment paragraph is treated as if it were shifted left by four bits. SI is a 16-bit segment index register that contains a relative offset from the segment paragraph specified in SR. What will be the actual memory address accessed if the contents of SR are 1234H and the contents of SI are 4392H?

   Answer:

   Since according to the description, $1234H : 4392H$ is the address of the destination location in main memory based on the segmentation scheme, the actual memory address in the linear address space is:

   $$1234H << 4 + 4392H = 12340H + 4392H = 166D2H$$

2. As a deadlock prevention strategy, the *hold-and-wait* condition may be prevented. If you are allowed to use semaphores, how would you use this strategy to regulate the requests for resources so that deadlock is prevented? Give a skeleton of the program in the same style as the first example regarding the two processes, P and Q, on 10/22's notes.

   Answer:

   To avoid circular wait by preventing the hold-and-wait condition, all resources required by a process may be allocated in the manner of "all at once" or "nothing at all". The scheme may be illustrated as follows:

```
Process P:
  ...
  wait(semaphore)
  Get A
  Get B
  ...
```

```
do_something()
...
Release A
Release B
signal(semaphore)
...
```

Forcing the processes to request for the various resources in a specific order is another way to prevent the hold-and-wait condition.

```
Process P:
  ...
  Get A
  Get B
  ...
  do_something()
  ...
  Release A
  Release B
  ...
```

```
Process Q:
  ...
  Get A
  Get B
  ...
  do_something_else()
  ...
  Release B
  Release A
  ...
```

3. Prove the correctness of Dekker's algorithm in the following aspects:

   (a) Prove that mutual exclusion is enforced.

   Proof 1:

   To show mutual exclusion is enforced in Dekker's algorithm, we prove, based on Figure 5.3 in the textbook, no any other process can get into its critical section after one of the processes is already in its critical section.

   Suppose process P_1 enters its critical section first and remains there. According to the algorithm, $flag[1]$ must be $true$.

   Since P_1 is the only one to get in, another process, say P_0, that wants to get in as well must be executing a statement outside its critical section. Since the $while$ loop is the only control structure that may block its access, we need only to discuss the case when P_0 is executing the $while$ statement. According to $P0()$, when $flag[1]$ is $true$, P_0 won't be permitted to exit the $while$ loop and thus enter the critical section, whatever $turn$ is.

   Hence, at most one process may be in its critical section at a time. Mutual exclusion is enforced indeed in Dekker's algorithm.

   Proof 2:

Suppose two processes, P_0 and P_1, are in their respective critical sections in the purpose of contradiction.

According to Dekker's algorithm in Figure 5.3 in the textbook, we know:

- For P_0, $flag[0]$ is set to $true$ and **then** $flag[1]$ is checked and confirmed to be $false$, which may be denoted by:

```
...
1  flag[0] = true
...
2  flag[1] = false
...
```

- For P_1, $flag[1]$ is set to $true$ and **then** $flag[0]$ is checked and confirmed to be $false$, which may be denoted by:

```
...
3  flag[1] = true
...
4  flag[0] = false
...
```

Suppose P_0 entered the critical section no later than P_1, which means $t_2 \leq t_4$, noting that the confirmation of the falsity of the condition in the outer $while$ loop is the last action a process takes before it enters the critical section.

Then we compare t_2 and t_3. Since once $flag[1]$ is set to $true$ at t_3, it will by no means become $false$, we know $t_2 < t_3$ must hold. We use $<$ here instead of $\leq$ because a location in main memory cannot be accessed simultaneously by multiple processes. Thus we know

$$t_1 < t_2 < t_3 < t_4$$

However once $flag[0]$ is set to $true$ at t_1, it will by no means become $false$ before P_0 finally exits the critical section, noting we suppose that both P_0 and P_1 are in the critical sections at the present time. Thus it is impossible to have checked $flag[0]$ to be $false$ at t_4, which according to our assumption has happened.

Thus a contradiction is drawn. We cannot assume two processes are in the critical sections at any moment, meaning at most one process may access the exclusive resources at one time. So mutual exclusion is enforced in Bekker's algorithm.

(b) Prove that a process requiring access to its critical section will not be delayed indefinitely. That is to show there is no starvation.

Proof:

To show there is no starvation in Dekker's algorithm, we prove, based on Figure 5.3 in the textbook, any process can eventually enter its critical section if it wants.

Suppose process P_1 wants to enter its critical section. Since the $while$ loop is the only control structure that may block its access, we simply suppose it is executing the $while$ statement.

- **If** $flag[0]$ **=** $false$:
  P_1 will surely fail in checking for $while\ (flag[0])$ and thus get into the critical section. One exception is P_1 may not be able to reach $while\ (flag[0])$, due to being trapped at $while\ (turn\ ==\ 0)$. If it is being blocked there, $flag[1]$ has been set to be $false$ to show courtesy. Thus P_1 will in no way present P_0 from accessing the exclusive resources, which will enable P_0 to enter the critical section and finally set $turn$ to 0 and $flag[0]$ to $false$. Thus P_1 may eventually get out of the loop of $while\ (turn\ ==\ 0)$.
  We need not discuss the case when P_0 is not running, because for P_1 to arrive at $while\ (turn\ ==\ 0)$, $flag[0]$ must have once been $true$ and P_0 must have been active, which shows $turn$ and $flag[1]$ have obtained or will have eventually the values giving the green light for P_1 to enter the critical section.
- **Or otherwise** $flag[0]$ **=** $true$:
  Similar to the ending part of the discussion in the first case, $turn$ and $flag[1]$ have been or will be eventually given 1 and $false$ respectively, which guarantees P_1 can exit the $while$ loop at last and get into the critical section.

In either case, P_1 will finally be able to enter its critical section. So due to the equivalence of all the individual processes, any process, if it wants, can eventually visit the exclusive resources. Hence no starvation is in Dekker's algorithm.

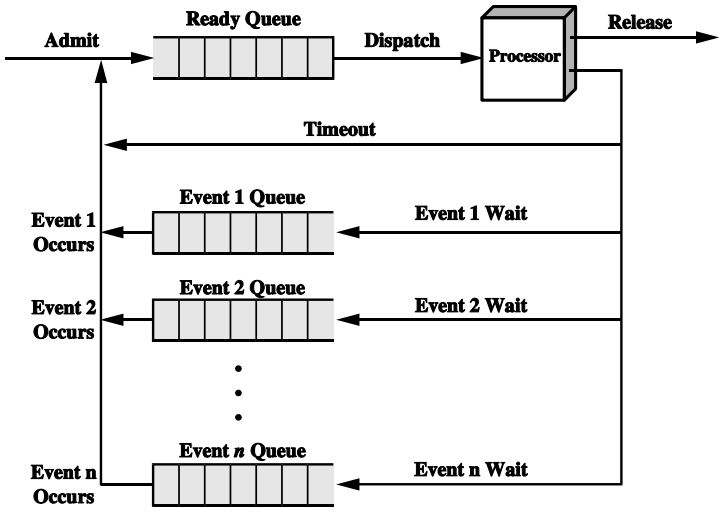
4. The following figure illustrates that a process may be blocked and placed into the corresponding event queue due to waiting for an event of a specific type, but it suggests that a process can only be in one event queue at a time.

   (a) Is it possible that you would want to allow a process to wait on more than one event at the same time? Provide an example.

   Answer:

   Yes, it is possible that a process waits on more than one event at the same time. For example, a process may need to transfer data from one device to another. In this case, it may request both devices at once and wait until both are available for use. Another example is that a network application may wait on multiple sockets until data packets arrive at any of them.

   (b) In that case, how would you modify the queuing structure of the figure to support this new feature? Give the definition of the structures in C/C++ and illustrate it in a picture.



Answer:

Obviously in the event queue model, at least 3 kinds of structures are needed: *process*, *event*, and *process-event pair*, which is defined for event queue nodes.

To enable a process to wait on multiple events at the same time, a chain may be constructed to track all the events that a process is expecting. The head pointer of the chain is stored in `struct process`.

The nodes in event queues thus need additional fields for the event chains they belong to: `struct pe_node *pred, *succ`.

Another issue that needs to be dealt with is that though a process may wait on more than one event at a time, either *all* the events are required for the process to become READY or *any* of them is sufficient to do so. Thus `count` is defined in `struct process`. In the first case, `count` is initialized to be the number of events on which the process is waiting, while in the second case, `count` is set to 1. Whenever an event occurs, `count` is first decreased by 1 and the process-event pair node is removed from both the corresponding event queue and event chain. Then `count` is checked if it becomes 0. If yes, the process is then sent to the READY process queue for dispatching; otherwise continues to wait on the rest of events.

```
struct pe_node {
  // for neighbors in the same event queue
  struct pe_node *next, *prev;

  // for neighbors in the event chain for the same process
  struct pe_node *pred, *succ;

  struct process *p;
  struct event *e;
}
```

```
struct process {
  PCB *pPCB;
  struct pe_node *head;
  int count;
}

struct event {
  char *desc;
  struct pe_node *head;
}
```

5. Do the followings regarding message passing.

   (a) Describe how mutual exclusion and synchronization are supported with message passing.

   Answer:

   (By default, `send` and `receive` primitives that we use for mutual exclusion and synchronization are respectively non-blocking and blocking.)

   To support *mutual exclusion*, a mailbox should be created first and initialized to contain a token message. Each process that is going to access critical resources has to invoke `receive` first to obtain the token before moving on. If permitted, when it exits the critical section, it again sends the token back to the mailbox so that other processes may enter their critical sections later on.

   To support *synchronization* (meaning that a process has to wait until another process is ready for some action), similarly a mail box is needed but is initially empty. The process that is supposed to wait should invoke `receive` and the process that is expected should invoke `send` indicating its availability.

   (b) Give the solution to the producer/consumer problem with an infinite buffer using message passing.

   Answer:

```
producer() {
  while (true) {
    product = produce();
    send(mayconsume, product);
  }
}

consumer() {
  while (true) {
```

```
    receive(mayconsume, product);
    consume(product);
  }
}

main() {
  create_mailbox(mayconsume);
  parbegin(producer, consumer);
}
```

(c) Give a fair solution to the barbershop problem using message passing in a different way from the solution given in the textbook. (*Hint: You may assign a unique number to each barber chair instead of each customer.*)

Answer:

```
void main() {
  create_mailbox(max_capacity);
  for (int i=0; i<20; i++)
    send(max_capacity, null);

  create_mailbox(sofa);
  for (int i=0; i<4; i++)
    send(sofa, null);

  create_mailbox(chair);
  for (int i=0; i<3; i++) {
    send(chair, i);
    create_mailbox(finished[i]);
  }
  create_mailbox(cust_ready);

  create_mailbox(coord);
  for (int i=0; i<3; i++)
    send(coord, null);

  create_mailbox(payment);
  create_mailbox(receipt);
}

void barber() {
  int chair_no;

  receive(cust_ready, chair_no);
  receive(coord, -);
  cut_hair();
  send(coord, null);
  send(finished[chair_no], null);
}

void customer() {
  int chair_no;

  receive(max_capacity, -);
  enter_shop();
  receive(sofa, -);
  sit_on_sofa();
  receive(chair, chair_no);
  get_up_from_sofa();
  send(sofa, null);
  sit_in_chair(chair_no);
  send(cust_ready, chair_no);
  receive(finished[chair_no], -);
  leave_chair();
  send(chair, chair_no);
  pay();
  send(payment, null);
  receive(receipt, -);
  exit_shop();
  send(max_capacity, null);
}
```

```
void cashier() {
  receive(payment, -);
  receive(coord, -);
  accept_pay();
  send(coord, null);
  send(receipt, null);
}
```

In the above solution, the hair-cut sessions are differentiated by the *chair NOs* instead of *customer IDs* used in the textbook. And with message passing, the interaction between a barber and a customer is always associated with the corresponding chair NO, which is transferred as a message between them.

6. Use semaphores to solve the following problem:

   You have been hired by Greenpeace Organization to help the environment. Because unscrupulous commercial interests have dangerously lowered the whale population, whales are having synchronization problems in finding a mate. The trick is that in order to have children, `three` whales are needed, one male, one female, and one to play matchmaker – literally, to push the other two whales together (I'm not making this up!). Your job is to write the three procedures `Male()`, `Female()`, and `Matchmaker()`. Each whale is represented by a separate process. A male whale calls `Male()`, which waits until there is a waiting female and matchmaker; similarly, a female whale must wait until a male whale and a matchmaker are present. Once all three are present, all three return.

Answer:

```
semaphore male = 0, female = 0;
semaphore male_start = 0, male_end = 0;
semaphore female_start = 0, female_end = 0;

Male() {
  signal(male);
  wait(male_start);
  ...
  wait(male_end);
}

Female() {
  signal(female);
  wait(female_start);
  ...
  wait(female_end);
}
```

```
Matchmaker() {
  wait(male);
  wait(female);
  signal(male_start);
  signal(female_start);
  match();
  signal(male_end);
  signal(female_end);
}
```

Note that this is an unfair solution, in which a matchmaker may send the `male_start` / `female_start` signal to another male/female whale rather than the one that has been allocated to it after `wait(male)` / `wait(female)`. The same problem happens when a `male_end` / `female_end` signal is sent out. These problems may be solved in the same way as the barbershop example in the text by using queues or the solution we give above in Question 5(c).