Week 11. RTOS – Concurrency

18-342: Fundamentals of Embedded Systems

Rajeev Gandhi
INI & ECE
Carnegie Mellon University

Carnegie Mellon

Electrical & Computer ENGINEERING

Outline

- Process concurrency
  - Software solutions
  - Hardware solutions
    - Atomic test and set instruction
    - Atomic swap instruction
  - Semaphores
  - Implementation
- Process synchronization
  - Producer-Consumer problem and solutions
- Quiz 3 on Thursday
  - Syllabus: post-midterm to the current lecture
  - Same rules as quizzes 1 and 2

What is the Critical-Section Problem?

- A number of processes share a common segment of code (or data), called the critical section
- When one process is executing its critical section, no other process must be allowed to execute its critical section
- Problem: Design a protocol for the processes to cooperate
- Correct solution must satisfy:
  - Mutual exclusion: No two processes may be simultaneously in their critical sections
  - Progress: No process running outside the critical section may block another process from entering the critical section
  - Bounded waiting: There is a bound on the number of times that other processes can get into the critical section after P makes a request to enter the critical section and before that request is granted

Solution 1 – Taking Turns

- Use a shared variable to keep track of whose turn it is
- If a process, Pi , is executing in its critical section, then no other process can be executing in its critical section
- Solution 1 (turn is initially set to 1)

process one

```
while(turn != 1)
   ; // loop till turn == 1
   x = x + 1;
   turn = 2;
```

process two

```
while (turn != 2)
      ; //loop till turn = =2
      x = x + 1;
      turn = 1;
```

- Hmmm…..what if Process 1 sets *turn = 2,* but Process 2 never enters its critical section?
- We have mutual exclusion, but does Process 1 make progress?

Solution 2 – Status Flags

- Have each process check to make sure no other process is in the critical section
- Use 2 shared variables (`P1inCrit` and `P2inCrit`)

initially, P1inCrit = P2inCrit = 0;

process one

```
while (1){
  while(P2inCrit == 1);
    P1inCrit = 1;
    x = x + 1;
    P1inCrit = 0;
}
```

process two

```
while (1) {
while (P1inCrit == 1);
      P2inCrit = 1;
    x = x + 1;
    P2inCrit = 0;
}
```

- Do you see any problems here?
  - The algorithm depends on the exact timing of the two processes

Solution 2 Does **not** Guarantee Mutual Exclusion

process one

```
while (1){
  while(P2inCrit == 1);
    P1inCrit = 1;
    x = x + 1;
    P1inCrit = 0;
}
```

process two

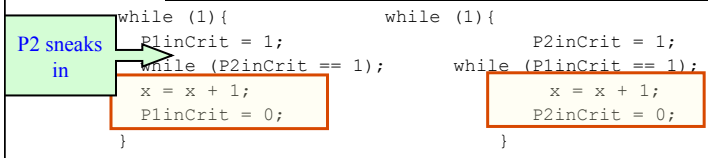
```
while (1){
  while (P1inCrit == 1);
    P2inCrit = 1;
    x = x + 1;
    P2inCrit = 0;
}
```

P2 sneaks in

	P1inCrit	P2inCrit	
Initially	0	0	
P1 checks P2inCrit	0	0	P1 jumps out of while loop
P2 checks P1inCrit	0	0	P2 jumps out of while loop
P1 sets P1inCrit	1	0	
P2 sets P2inCrit	1	1	
P1 enters crit. section	1	1	
P2 enters crit. Section	1	1	

Solution 3: Enter the Critical Section *First*

- Set your own flag before testing the other one



	P1inCrit	P2inCrit
Initially	0	0
P1 sets P1inCrit	1	0
P2 sets P2inCrit	1	1
P1 checks P2inCr	1	1
P2 checks P1inCrit	1	1

- Each process waits indefinitely for the other

Deadlock - when the processes can do no more useful work

Peterson's Solution – Take Turns & Use Status Flags

process one

```
while (1){
 P1inCrit = 1;
 turn=2;
 while (P2inCrit == 1 &&
       turn==2){};

  x = x + 1;
  P1inCrit = 0;
}
```

process two

```
while (1){
  P2inCrit = 1;
  turn=1;
  while (P1inCrit == 1 &&
              turn==1){};

  x = x + 1;
 P2inCrit = 0;
}
```

- Initially, `turn` = 1 and `P1inCrit` = `P2inCrit` = 0;
- Ensures Progress, Mutual Exclusions, Bounded Waiting

Hardware Solutions to the Critical Section Problem

- Disabling (and enabling) interrupts
  - Cannot allow user programs to disable interrupts
- Special Instructions
  - `TAS` - `Test and Set` instruction
    - Both of the following steps are executed **atomically**
      - TEST the operand and set the CPU status flags so that they reflect whether it is zero or non-zero
      - Set the operand, so that it is non-zero
- Example
  - Initialize `lockbyte` to 0

```
Process 1                          Process 2
LOOP:  TAS lockbyte          LOOP:  TAS lockbyte
       BNZ LOOP                     BNZ LOOP
         critical section             critical section
       CLR lockbyte                 CLR lockbyte
```

(TAS is not an ARM instruction)

Called a **busy-wait** (or a **spin-lock**)

Hardware Solutions (contd)

- Alternatively, hardware can provide a special instruction to swap the contents of two words **atomically**

```
void Swap(boolean *a, boolean *b) {
boolean temp = *a;
*a = *b;
*b = temp;
}
```

```
Process 1                          Process 2
while(1) {                         while(1) {
   key1 = true;                     key2 = true;
   while(key1 == true)              while(key2 == true)
        Swap(&lock, &key1);           Swap(&lock, &key2);
   x=x+1;                           x=x+1;
   lock = false;                    lock = false;
};                                 };
```

Initialize `lock=false`

Hardware Solutions (contd)

- Atomic SWP instruction on the ARM processor

- `SWP<cond> {B} Rd, Rm, [Rn]`

  Equivalent to

  ```
  temp = Mem[Rn]
  Mem[Rn] = Rm
  Rd = temp
  ```

- `SWP` combines a load and a store in a single, atomic operation

  ```
  SWPB   r1, r1,[r0]
  ```

  Atomically swaps the contents of memory location pointed by `r0` with contents (bits 0-7) of register `r1`

Critical Section For More Than Two Processes

- How do you extend previous algorithm for more than two processes?
- The **Bakery Algorithm**
  - On arrival at a bakery, the customer picks a token with a number and waits until called
  - The baker serves the customer waiting with the lowest number
    - Same applies today at AAA and DMV ;-)

Semaphores

- **Simplest** form of concurrent programming
- Peterson's/Dekker's algorithm is difficult to extend to 3 or more processes
- Previous solutions can be complicated for application programmers to use
- Semaphores are a much easier mechanism to use

Semaphores

- **Semaphore** - an integer variable (> 0) that normally can take on only non-zero values
- Only three operations can be performed on a semaphore - **all operations are atomic**
  - `init(s, #)`
    - sets semaphore, `s`, to an initial value #
  - `wait(s)` or `P(s)`
    - if `s > 0`, then `s = s - 1;`
    - else suspend the process that called `wait`
  - `signal(s)` or `V(s)`
    - `s = s + 1;`
    - if some process P has been suspended by a previous `wait(s)`, wake up process P
      - *normally*, the process waiting the longest gets woken up

Uses: Mutual Exclusion with Semaphores

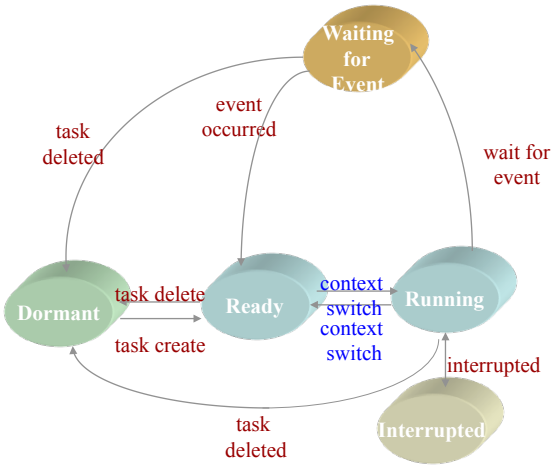
```
process one                    process two
while (1){                     while (1){
  wait(s);                       wait(s);
  x = x + 1;                     x = x + 1;
  signal(s);                     signal(s);
}                              }
```

- initially, `s = 1` (this is called a **binary semaphore or a mutex**)

Implementation

- Can be implemented by disabling (and enabling) interrupts
  - Disabling interrupts occurs at the OS level ➔ can ensure proper usage
  - Applications can only modify semaphores using the `init`, `wait` and `signal` operations
- Can also use any of the previously mentioned methods to ensure atomic operation
  - Disadvantage?
- What is the difference between blocking and busy waiting (or spin locks)?

Busy Waiting Vs Blocking



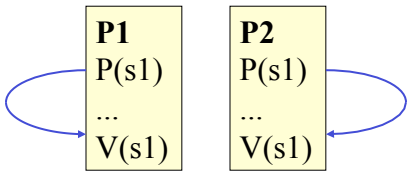

```
Assume P1inCrit=1 and
turn = 1 then Process 2
busy  waits on the
following while loop

while (P1inCrit == 1 &&
       turn==1) {};
```

• In busy waiting, the process doing busy waiting transitions back and forth between "running" state and "ready" state until it acquires the lock
• A blocked process is moved from running state to "waiting for event" state and is put into "ready" state after the lock has been released

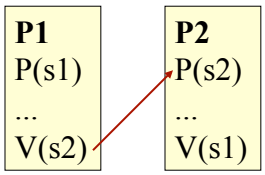
Types of Synchronization with Semaphores

- There are 2 basic types of synchronization
  - Mutex – where semaphores serve to protect a critical section (shared data/code)
  - Barrier synchronization – where semaphores are used sort of as an inter-task communication facility to "wake up" other waiting processes
- Remember P=wait and V=signal
- Here, s1 and s2 are semaphores



Mutex

Initial value of s1 = 1



Barrier Synchronization

Initial value of s1 = 1, s2 = 0

Producer-Consumer Problem

- In this problem, there are two kinds of processes
  - Producer processes: generate data
  - Consumer processes: consume data
- Ordering constraints
  - Data needs to be produced before it can be consumed
  - Can't consume more data than has been produced
- How do you write an application program that involves these kind of processes with the constraints?
  - OS provides an API to create/use semaphores but does not enforce the process ordering required by producer-consumer
  - Application code needs to be written to enforce the ordering constraint using semaphores

The Producer-Consumer Problem

- One process produces data, the other consumes it
  - (e.g., I/O from keyboard to terminal)

```
producer(){
  while(1){
    produce();
    appendToBuffer();
    signal(itemReady);
  }
}
```

```
consumer(){
  while(1){
    wait(itemReady);
    takeFromBuffer();
    consume();
  }
}
```

Initially, `itemReady = 0;`

Another Producer/Consumer

- What if both `appendToBuffer` and `takeFromBuffer` **cannot** overlap in execution?
  - For example, if buffer is a shared link list between producer and consumer?
  - Or, multiple producers and consumers

```
producer() {
  while(1){
    produce();
    wait(mutex);
    appendToBuffer();
    signal(mutex);
    signal(itemReady);
  }
}
```

```
consumer() {
  while(1){
  wait(itemReady);
  wait(mutex);
  takeFromBuffer();
  signal(mutex);
  consume();
  }
}
```

- Initially, `mutex = 1, itemReady = 0;`

Bounded Buffer Problem

- Assume a single buffer of fixed size
  - Consumer blocks (sleeps) when buffer is empty
  - Producer blocks (sleeps) when the buffer is full

```
producer() {
   while(1) {
      produce();
      wait(spacesLeft);
      wait(mutex);
      appendToBuffer();
      signal(mutex);
      signal(itemReady);
   }
}
```

```
consumer() {
    while(1){
      wait(itemReady);
      wait(mutex);
      takeFromBuffer();
      signal(mutex);
      signal(spacesLeft);
      consume();
    }
}
```

- Initially, `mutex = 1, itemReady = 0; spacesLeft = sizeOfBuffer;`

Deadlocks

- When a program is written carelessly, it may cause a... **deadlock**!!!

P1::	**P2::**
P(s1);	P(s2);
P(s2);	P(s1);
...	...
V(s2);	V(s1);
V(s1);	V(s2);

Deadlocks

- Several processes "executing" at the same time, and one is waiting (blocked) for (by) another, which in turn is waiting for another, which in turn... which in turn is waiting for the first.
  - No process can finish, since they are all waiting for something
- The difference between deadlock and starvation of a process is that
  - In deadlock, a process must be able to acquire a resource at first, and then go into deadlock.
  - In starvation, the request may be deferred infinitely
    - the resource may be in use for an infinite amount of time

Resources

- What could be a resource?
  - Hardware devices e.g. printer, tape drive needed by a process to do useful work
  - Software resource e.g. mutex
- Processes utilize resources in the following sequence
  - Request (must wait if request is not granted)
  - Use
  - Release
- How do we get into a deadlock
  - Consider a system with one printer and one tape drive
  - Process 1 requests printer and the request is granted
  - Process 2 requests tape drive and the request is granted
  - Process 1 also needs tape drive to finish its job
    - Must wait for Process 2 to release tape drive
  - Process 2 also needs printer to finish its job
    - Must wait for Process 1 to release tape drive
  - Deadlock!

Dining Philosophers

- *n* philosophers *P0, P1, ..., Pn-1*
- To the left of each philosopher is a fork, and in the center is a bowl of spaghetti
- A philosopher was expected to spend most of his time thinking; but when he felt hungry, he needed to pick up both the left fork and right fork to eat the spaghetti
- When he was done with his meal, he puts down both sticks and continues thinking
- A fork can be used by only one philosopher at a time
  - If a neighboring philosopher wants it, he has to wait until the fork is available again

Dining Philosophers

- Recap of problem statement
  - N Philosophers sitting at a round table
  - A fork is placed between each philosopher and its neighbor
  - Each philosopher must have both forks to eat spaghetti
  - Neighbors cannot eat simultaneously
  - Philosophers alternate between thinking and eating
- Each philosopher runs following code:

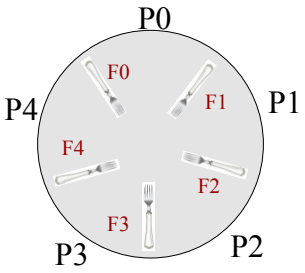
```
while (1) {
  think();
  take_left_fork(i); // grabs left fork
  take_right_fork(i);
  eat();
  release_left_fork(i); // releases left fork
  release_right_fork(i);
}
```

Dining Philosophers Solution #1

- Two neighbors cannot use a fork at same time
- Must test if fork is there and grab it atomically
  - Represent each fork with a semaphore
  - Grab right fork, then left fork
- Code for 5 philosophers:

```
sem_t fork[5]; // Initialize each to 1
take_forks(int i) {
  wait(&fork[i]);
  wait(&fork[(i+1)%5]);
}
release_forks(int i) {
  signal(&fork[i]);
  signal(&fork[(i+1)%5]);
}
```

- Do you see a problem here?

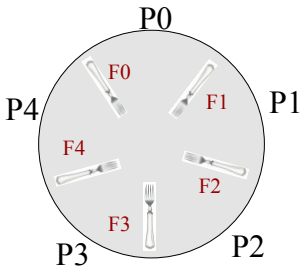


Dining Philosophers Solution #2

- Approach
  - Odd numbered philosopher: grab lower-numbered fork first, then higher-numbered one
  - Even numbered philosopher: grab higher-numbered fork first, then lower-numbered one
- Code for 5 philosophers:

```
sem_t fork[5]; // Initialize to 1
take_forks(int i) {
  if (i%2 == 1) {
        wait(&fork[i]);
        wait(&fork[i+1]);
  } else {
        wait(&fork[i+1]);
        wait(&fork[i]);
}
```

- Do you see a problem here?



Dining Philosophers: Getting it Right

- Force all the philosophers to obey certain rules
  - Each philosopher must check whether both forks are available and pick both of them atomically (if they are available)
  - If a philosopher is hungry but one or both forks are not available then the philosopher must suspend until the forks become available
- Let's go with 5 philosophers again
- Introduce state variable for each philosopher *i*

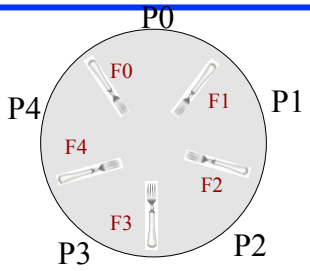
  `state[i] = THINKING, HUNGRY, or EATING`
- Need to ensure
  - No two adjacent philosophers eat simultaneously

    ```
    for all i
       !(state[i]==EATING && state[i+1%5]==EATING)
    ```
  - NOT the case that {a philosopher is hungry *and* his neighbors are not eating}

    ```
    for all I
        !(state[i]==HUNGRY &&
          (state[i+4%5]!=EATING && state[i+1%5]!=EATING))
    ```

Putting All the Pieces Together

```
sem_t mayEat[5]; // how should we initialize this?
sem_t mutex; // how should we initialize this?
int state[5] = {THINKING};
take_forks(int i) {
    wait(&mutex); // enter critical section
    state[i] = HUNGRY;
    testSafetyAndLiveness(i); // check if I can run
    signal(&mutex); // exit critical section
    wait(&mayEat[i]);
}
release_forks(int i) {
    wait(&mutex); // enter critical section
    state[i] = THINKING;
    testSafetyAndLiveness (i+1 %5); // check if my neighbor can run now
    testSafetyAndLiveness(i+4 %5);
    signal(&mutex); // exit critical section
}
testSafetyAndLiveness(int i) {
    if (state[i]==HUNGRY && state[i+4%5]!=EATING&&state[i+1%5]!=EATING) {
          state[i] = EATING;
          signal(&mayEat[i]);
    }
}
```



Necessary Conditions for a Deadlock

- Formally: (***all*** of these conditions ***must*** hold for deadlock to occur)
  - **Mutual exclusion**: some resource is non-sharable, (eg, a printer)
  - **Hold and wait**: there must exist a process holding for a resource (eg, a printer) and waiting for another resource (eg, a plotter)
  - **No preemption**: the system will not preempt the resources in contention
  - **Circular wait**: there must exist a circular chain of processes.
    - One is holding a resource and waiting for the next process

Deadlock Handling

- **Prevention**: structure the system in such a way as to avoid deadlocks (i.e., in a way to avoid one of the conditions above).
  - This is done in the design phase: design a system and ensure that there is no deadlock in the system
- **Avoidance**: does not make deadlock impossible (as in prevention)
  - Instead, it rejects requests that cause deadlocks by examining the requests before granting the resources. If there will be a deadlock, reject the request
- **Detection and Recovery**: after the deadlock has been detected, break one of the 4 conditions above. The mechanisms of deadlock detection and deadlock recovery are very much tied to each other

Deadlock Prevention

- Ensure one of the four necessary condition is never satisfied
  - Allow all resources to be shared (so prevent mutual exclusion)
    - Some resources are inherently non-sharable
  - Don't allow hold and wait i.e. force processes to either acquire all the needed resources or to release the acquired (currently held) resources if one or more requests are not granted
    - Starvation of processes that need many resources
  - Allow preemption of resources
  - Enforce total ordering in resource acquisition process and require each process to request resources in an increasing order
    - Example: Printer order=3, Tape drive order = 7
      - If process 1 needs printer and tape drive in any order it must request printer first and then tape drive

Can cause deadlock

```
P1::        P2::
P(s1);      P(s2);
P(s2);      P(s1);
...         ...
V(s2);      V(s1);
V(s1);      V(s2);
```

Prevents deadlock

```
P1::        P2::
P(s1);      P(s1);
P(s2);      P(s2);
...         ...
V(s2);      V(s1);
V(s1);      V(s2);
```

Deadlock Avoidance

- All processes need to declare in advance all the resources that they will need
- Given all the resources all the processes will need, the system can decide for each resource request whether or not a process should wait for the request to be granted
- **Banker's Algorithm**
- Based on how a bank would work with several clients that borrow and invest money
  - It differentiates between **safe** state (resources can be allocated to the processes in such a way that there will be no deadlock) and **unsafe states** (the opposite)
  - The OS will refuse requests for resources from processes if the request will take the system to an unsafe state.
  - If the system is always in a safe state, there will never be a deadlock.
  - When a request arrives, the Banker's algorithm checks whether there will be enough resources (money) to cover all the money to be returned to the clients in case they decide to withdraw.

Banker's Algorithm

- A, B, C, D are four bank customers
  - Each customer has been given a line of credit
  - Each customer may need all their credit units to do their job
    - Customers may not request their maximum credit units all at once
    - After using their credit, customers repay the loan
- Bank has 10 dollars even though 22 dollars are needed to satisfy each customer's request in the worst case
  - Customers must wait if their request cannot be satisfied
  - Deadlock occurs when all customers have to wait for each other
- Check whether the following bank states are safe

	Has	Max
A	0	6
B	0	5
C	0	4
D	0	7

Free: 10

	Has	Max
A	1	6
B	1	5
C	2	4
D	4	7

Free: 2

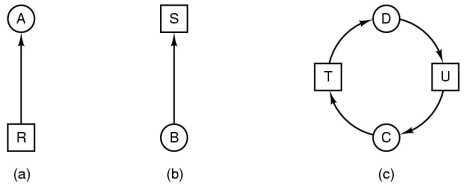
	Has	Max
A	1	6
B	2	5
C	2	4
D	4	7

Free: 1

Source: Modern Operating Systems by A. Tanenbaum

Deadlock Detection

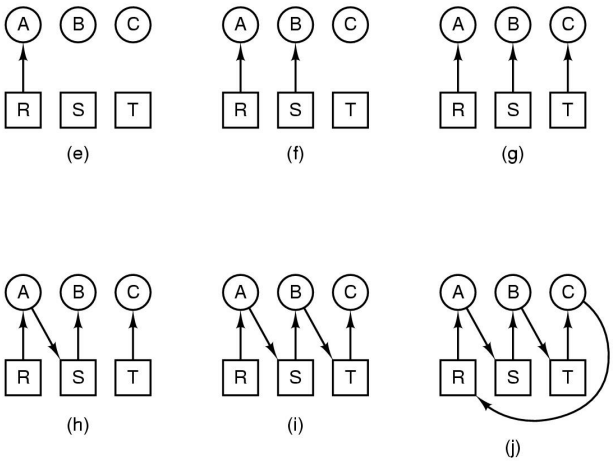
- Graph models for deadlock detection
- If each resource has a single instance in a system, a *directed* graph model can directly tell you if there is a deadlock in the system
  - Called resource allocation graph



  - resource R assigned to process A
  - process B is requesting/waiting for resource S
  - process C and D are in deadlock over resources T and U

Source: Modern Operating Systems by A. Tanenbaum

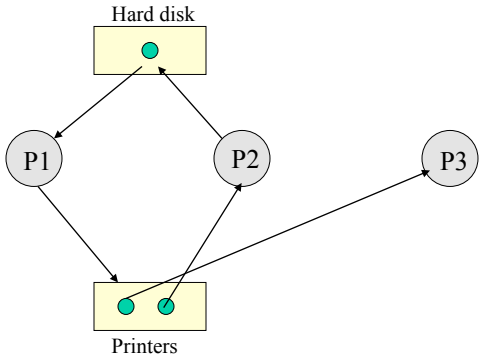
Deadlock detection with one resource of each type



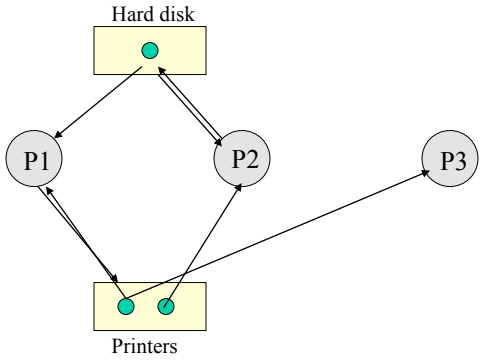
Source: Modern Operating Systems by A. Tanenbaum

Deadlock detection with multiple resources

- Assume a system with 2 printers and 1 hard disk – the resource allocation graph is shown below
  - Is there a deadlock?



Deadlock detection with multiple resources



With multiple resources of each type
If resource allocation graph does not have a cycle then the system is not in deadlock state.
If resource allocation graph has a cycle then system *may* or may not be in deadlock